

AD-A037 640

COMPUTER SCIENCES CORP FALLS CHURCH VA
DOD PROGRAM FOR SOFTWARE COMMONALITY HIGH ORDER LANGUAGE WORKIN--ETC(U)
1977

F/G 9/2

N00039-75-C-0289

NL

UNCLASSIFIED

| OF 5
ADA037640



ADA 037640

Copy No. _____

5.3.7

DOD PROGRAM FOR SOFTWARE COMMONALITY
HIGH ORDER LANGUAGE WORKING GROUP

CANDIDATE LANGUAGES EVALUATION AND
RECOMMENDATIONS REPORT.

1977

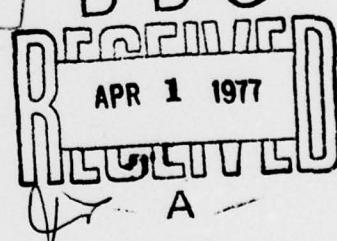
443P

Prepared for

NAVAL ELECTRONICS SYSTEMS COMMAND
Washington, D.C.

Under
CONTRACT NO0039-75-C-0289

15



COMPUTER SCIENCES CORPORATION

6565 Arlington Boulevard

Falls Church, Virginia 22046

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Major Offices and Facilities Throughout the World

405 717

PREFACE

The Department of Defense (DoD) High Order Language (HOL) Program has as a primary objective to establish a minimum number of HOLs for use throughout DoD. This volume, containing comparisons of:

CMS-2 ✓	JOVIAL J3B ✓
CORAL 66 ✓	JOVIAL J73 ✓
CS-4 ✓	PEARL ✓
EUCLID ✓	SPL/1

against the DoD language requirements document [✓]Tinman, ✓ represents Computer Sciences Corporation's contribution to DoD's goal. This work was performed under contract N00039-75-C-0289 and was monitored by Mr. Robert I. Kahane (ELEX 03E) Washington, D. C.

ACCESSION FOR	
NTIS	White Section <input checked="" type="checkbox"/>
DDC	Buff Section <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION.....	
BY.....	
DISTRIBUTION/AVAILABILITY CODES	
DIST.	AVAIL. AND/or SPECIAL
A	B

TABLE OF CONTENTS

<u>Preface</u>	ii
<u>Section 1 - Introduction.</u>	1-1
<u>Section 2 - Tinman Discussion.</u>	2-1
<u>Section 3 - Comparative Evaluations</u>	3-1
<u>Section 4 - Discussion and Summary</u>	4-1

SECTION I - INTRODUCTION

Like most large computer users, the Department of Defense (DoD) has been plagued with a proliferation of High Order Languages (HOL) and incompatible systems serving the "same" language. The DoD's problems are, in principle, no different from the rest of the computer user community; they are simply larger as is the use of computers. Further, DoD systems are often individually very large and very long lived. Because of the intimate integration of the computer resources with the rest of a large defense system, it has not been possible to change computer subsystems with the frequency which might characterize a commercial operation. As a result, maintenance of the computer system is both long-term and dynamic, and maintenance in many cases involves modification of the system to respond to new threats. Defense systems are often composed of interacting but independently developed subsystems, sometimes brought into existence over a period of years, all of which must be served by a common but evolving hardware base. In such an environment, the DoD finds itself spending an increasingly larger fraction of its systems resources on software. HOL commonality and the resulting flexibility would provide a powerful tool for reducing the high cost of software in DoD.

With each of the Military Departments studying the problem and making proposals for common languages, it was clear that the greatest benefit could be reaped by providing languages common across the DoD. In January of 1975, a DoD High Order Language Working Group (HOLWG) was chartered by DDR&E with representatives from the Military Departments to investigate the requirements and specifications for programming language commonality, to compare these with existing approaches, and to recommend adoption or implementation of the necessary common languages or approaches. Until the matter is resolved, the DoD will not support the further implementation of new HOL in R&D programs.

The first task of the HOLWG was to formulate a set of requirements consistent with the levies of the Military Departments. Requirements were solicited from as broad

a base as possible, to be prioritized later as required. Inquiries were not restricted to those programs presently using HOLs; rather, a major thrust of the effort was to provide HOLs to meet the requirements of those who are now constrained to use an assembly language for lack of a suitable HOL.

It has been impossible to define rigorously the exact level requirements desired; therefore, a "Strawman" of preliminary requirements was established to define this level by illustration. The "Strawman" was not intended to be complete or consistent, rather it was deliberately provocative to elicit the widest possible comment. It was forwarded to the Military Departments, and by them to their various operating agencies. In addition, it was distributed to other Government agencies, the academic community, and industry; through industry organizations and military contractors; and by direct inquiry. A number of individuals and organizations have had an opportunity to examine this document and provide inputs. The bulk of such comments were positive and useful.

The results of four months of such input were put together in a more concrete form; one which could then be representative of a fairly complete set of requirements, although still a tentative set. This document was called the "Woodenman," and it too was distributed widely. It provided a more rigorous framework for specific comments. On the basis of all inputs and the official responses from each of the Military Departments, a more complete set of requirements has evolved and is called "Tinman." This document represents a set of requirements for HOL computer programming language consistent with the input from the Military Departments.

The next task of the HOLWG was an investigation and comparison of existing HOLs to the DoD HOLWG's language requirement document, "Tinman."

A candidate set of HOLs were chosen and contracted to various industry, individual, and university groups for evaluation. The effort represented by this volume, is Computer Sciences Corporation's contribution to the task of establishing a minimum number of HOLs for use throughout DoD. Each contractor investigated and compared selected languages to the DoD HOLWGs language requirement document "Set of Criteria

and Needed Characteristics for a Common DoD High Order Language, Tinman." Specifically, each contractor performed the following tasks:

- (1) For each language characteristic listed in "Tinman," the contractor will determine the "degree" of compliance of each of the candidate languages.
 - (a) If the requirement is "met," the contractor will demonstrate how this is so (e.g., the presence or absence of a particular feature, computer program, etc.). Additionally, the contractor must show that this solution does not conflict with any other DoD HOL requirement.
 - (b) For requirements which are "not met" or only partially met, the contractor will provide:
 1. An analysis of why the language does not fulfill or only partially fulfills the requirement.
 2. The scope of modifications that would be necessary to bring a language into compliance, and their impact on other language features.
 3. The impact of such modifications on implementations
 - (c) If the requirement is such that it is not addressed in the provided language documentation (e.g., the requirement is strictly dependent on the operating system, libraries or is only a management consideration), then the contractor will so indicate.
- (2) Upon completion of (1) the contractor will then identify features of the language which are not needed to satisfy the requirements with his recommendation as to whether those features should be kept or possibly eliminated.

SECTION 2 - TINMAN DISCUSSION

COMMENTS ON
TINMAN
REQUIREMENTS

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

The following is a collection of recommendations for improvements in the Tinman requirements. The comments do not address questions of interpretation or obvious omissions or wording problems, except where such omissions or problems cause the intent of a Tinman requirement to be unclear.

This report has one section on the Tinman as a whole, and a number of sections on individual requirements. Not all of the requirements are commented on, many of them being so clearly reflections of currently accepted language design philosophy that no comment is needed.

The version of the Tinman requirements addressed in this report is the one dated June, 1976.

The following comments and criticisms are intended constructively. We trust that they will be received in that way.

The principal criticism of Tinman as a whole is that it needs to be more carefully written. We say this while recognizing that it obviously has received a great deal of care and work. However, adherence to a few rules (which are, unfortunately, very laborious to follow) will result in the next version being much easier to understand. They are: (1) Be certain that all terms are clearly understood. (2) Use terms consistently. (3) Avoid using slight variations of terminology for reasons of style. The following paragraphs are short elaborations of these points. If any of the authors of Tinman find them to be so elementary as to be insulting, we apologize, but we think that the points should be made.

Assuring that all terms are clearly understood is quite difficult in a document of the nature of Tinman, because the simple expedient of defining all terms is not available. For example, it is clear that the authors of Tinman intended for the concept of type to be defined implicitly. In most cases, however, explicit definition is appropriate. In requirement P2 the word equivalence is used. In mathematics this word must be defined in each context, but no definition is given in Tinman and, as a result exactly what was intended in this requirement is not clear. In addition, there are a number of computer terms which the authors apparently thought would be clear to any professionals in the field, but which are open to interpretation (e.g., conformable arrays in requirement B7 and free union in requirement E7).

The inconsistent use of terms is universally recognized as something to be avoided, but it is very difficult to do without some sort of mechanical assist, such as a kwic index. Even with such an assist the job is tiresome. In requirement D5 it appears that the word range is being used in the sense of "set of", a meaning different from that used elsewhere in Tinman. The result is uncertainty as to the exact intent of this requirement.

Requirement A2

We maintain that a bit string type is necessary (see the comments on B10).

It is not clear from the statement of this requirement if literally a character type is being called for or if character type is an ellipsis for character string type. Given a character type and the definitional mechanisms called for in the Tinman, the common character string capabilities can be achieved. We believe, however, that the character string type is useful enough to warrant its inclusion as a compiler-defined type. Full-blown variable length strings are probably not needed. For most applications variable length strings with maximum lengths known at compile time suffice. Indeed, for most application strings with fixed length (fixed at compile time) are probably adequate.

We are confused by the mention of some type generators in both requirements A2 and E6. If this was intentional, then we have missed the point. It appears that requirement A2 should call for the "basic" types (integer, real, Boolean, character, character string, and bit string) and requirement E6 should cover the type generators (records, arrays, enumeration types, and powersets).

Requirement A4

To treat fixed point numbers as exact quantities often results in a great deal of run-time overhead. Usually multiple precision integer arithmetic is involved. If the fixed point capability is elaborate enough, full-blown multiple precision rational arithmetic could be necessary. Based on CSC's experience with Navy embedded computer systems (for example, NTDS), it is doubtful that the user community would accept such overhead.

Based on this same experience, we think that something slightly different is intended: Whenever two fixed point quantities are added or subtracted, any shifting necessary to line up radix points must be left shifts. That is, the usual floating point discipline of a right shift of one operand, discarding bits which correspond to "invisible" error bits of the other operand, is unacceptable; the other operand is considered not to have any "invisible" error bits -- it is exact. CMS-2 compilers perform their fixed point additions and subtractions in this manner for precisely this reason, and at the strong insistence of their user community.

Requirement A5

It is not clear if this requirement is for an ability to specify character sets or collating sequences -- two quite distinct concepts.

TINMAN
Requirement A5

?

Both of these concepts are also distinct from that of an enumeration type, so to be able to specify either using the notation of enumeration types is a conflict with requirement H10.

The remainder of our comments are based on the assumption that the subject is character sets.

In our view, the character set used is a property of the internal representation of the value, not of the value itself or of the operations available for it. Hence a character set is not a type, and which set is used depends on the containing variable (see also the comments on B1). If two character or character string variables have different representation -- different character sets -- then implicit, automatic conversion should occur when the value of one is copied to the other. Conversion might not always be possible, because a character may appear in only one of the character sets, but the same is true for numeric variables having different ranges, and the Tinman explicitly prohibits ranges from determining types (requirement D4). Explicit character set conversion operators are probably needed also, especially for literal values.

Requirement A6

Why must the lower subscript bound be determinable at compile time? If the user really needs a dynamic lower bound, this rule must make his code more awkward, harder to read, and a bit harder to optimize.

The statement that the default lower bound (our interpretation of preferable lower bound) is zero is arguable, particularly when one of the justifications is that it contributes to clarity. The great majority of applications of arrays are "natural" ones -- those which have a concept of a first element, a second element, etc., and a last element, and these should be represented by 1, 2, ..., and the array size, not 0, 1, and one less than the array size. Arrays which have a lower subscript bound different from 1 arise, for the most part, from specialized mathematical formulas, which probably play a very minor role in embedded computer applications in the Department of Defense.

Even the requirement that the values be contiguous might be unduly harsh. It is simple to implement non-contiguous subscript values without the expense of garbage collection; the compiler can simply construct a mapping function, which a programmer who needs such a capability would otherwise have to do himself. Requirement J1 prohibits this general case from affecting the efficiency of the ordinary, contiguous subscript, case. The only question, of course, is whether there is enough need to justify the inclusion of such a feature.

Requirement B2

The text of this requirement contains the sentence: "For floating point numbers identity will be defined as the same within the specified (minimum) precision." This cannot be done efficiently on binary machines, assuming that the precision specification is stated in decimal. Furthermore, ignoring any questions of efficiency, to build such a predicate into a language is dangerous because it would tend to encourage programmers to gloss over the difficulties of working with floating point data. The initial accuracy of floating point data is only one consideration when a comparison for equality is performed, and it may not even be the most important. The build-up of error in intermediate calculations must be taken into account, usually in an ad hoc fashion in each use.

Requirement B5

Avoiding truncation of the most significant digits implies run time checks. We recommend that the programmer be able to avoid this overhead (see also the comments on F7).

Requirement B6

This requirement specifies that "and" and "or" on scalars will be evaluated in short circuit mode. The intent of the qualification on scalars is unclear.

In addition, specification of short circuit mode conflicts directly with Requirement C1. We believe that short circuit mode evaluation, if provided, requires special syntax. It is hard to see how the given example improves clarity and maintainability. Dependencies should be clearly shown; for example:

if I < 7 then A(I) > X else false

or (using our own syntax)

if I < 7 (and A(I) > X).

A maintenance programmer cannot miss the intent here, as he easily can if a simple Boolean expression is used.

Hence we recommend that the second sentence of the requirement be replaced by:

Short circuit mode evaluation will be guaranteed only if special syntax, clearly showing the dependencies, is used. Otherwise, the programmer

will not be able to count on short circuit mode evaluation, or to assume that it will not occur.

The last clause of course means that the programmer can not depend on every function in an expression actually being called. If a function must be called, it can be put into a separate statement.

Requirement B8

This requirement does not explicitly forbid "mixed-mode" arithmetic, although that seems to be in the spirit of the requirement. "Mixed-mode" can be defined without resorting to implicit conversions by defining the arithmetic operators to depend on the type of their operands as well as their values. Of course, everyone knows that an implicit conversion is performed prior to execution of the operation, but if the conversion is done by the firmware it becomes difficult to argue that anything shady is going on. If a prohibition is intended, it should be explicitly stated.

With regard to the question of whether or not "mixed-mode" arithmetic should be prohibited, it becomes difficult to argue against when no information is lost during the conversion; for example, when an integer value is converted to fixed point or floating point, a fixed point value is converted to floating point, or a value of some type representing a real number is converted to the equivalent complex number. Indeed, many languages which have fairly strict type-checking allow something like this for constants at least through the device of specifying numeric constants to be typeless, specifying only a value -- not a type. It is probably best to specify that those conversions which do not lose any information (spelled out in detail, of course) can be performed implicitly.

If multiple character sets are to be supported, as indicated by requirement A5, the question of implicit conversion between character sets should be addressed. Again, such conversions should probably be permitted in any case in which no loss of information can occur.

Many programmers find conversions between integer and Boolean and between enumeration types and integers useful. We recommend that they be included.

Requirement B10

We recommend that something analogous to the FORTRAN FORMAT-controlled conversion also be required, both for I/O and internal conversions. In addition, we recommend that it be possible to copy a binary bit string into a variable of any type and vice versa. This too would be provided not only for I/O -- that is, it should be possible to

move a value of type bit string into any variable, without type conversion, and vice versa. This is sometimes the only efficient way to move data between machines, and may be the only way to break apart values whose structure and type is dynamically determined. It is, of course, machine and representation dependent, and allows escape from type checking. Requirement J3 states that machine dependence shall be possible. Type consistency rules, in our view, should not be regarded as a system of laws that the programmer cannot violate. Instead, they are a way of using redundancy to ensure that the programmer means what he writes. However, the programmer knows more than the compiler, and may not always be able to stay within the constraints laid down -- he must be allowed to violate them, if he states clearly that such is his intent. This is analogous to providing structured programming control structures, but keeping the goto for use where necessary (see the comments on C2).

Requirement C1

To write two function references, with both of the functions having side effects, and then expect the side effects to occur in a specific order is a terrible programming practice. Yet the effect of this requirement will be to give it an aura of legitimacy.

Also, in practice, this rule restricts optimization quite a bit. Furthermore, the meaning is not clear for some cases (e.g., embedded assignment). A better requirement would be:

"There shall be no rule in the language specifying that every argument of an expression is to be evaluated when the expression is evaluated, nor making the value of an expression dependent on the order of evaluation of its arguments, except where such dependency is clearly specified as part of the definition of an operator."

This rule can even be enforced by the compiler, in one sense, by having it generate "side-effect prone" operands (function references, embedded assignments, etc.) in a random order, using as its randomizer some datum over which the programmer has no control (e.g., some low-order bit of the hardware clock), which would presumably bring out any incorrect dependency on the order of occurrence of side effects early in the debugging stage. We know of no compiler which has used such a technique.

Requirement C2

"Few levels" should be more precisely defined. One level, as in APL, seems plainly to few. Wirth now believes that even Pascal has too few levels.

Requirement D2

The requirements spelled out here are good, in spite of any possible disagreements from compiler writers. Another one needs to be added: Alternative literal forms for the same quantity will have the same value. For example (using FORTRAN notation), a comparison of 1.1 and 0.11E1 should always give an equal result. Some compiler writers might well complain about this too.

Requirement D3

Some comment should be made about the initialization of variables shared between independently compiled programs -- e.g., in COMMON. We believe that for any named block of such variables, initialization should be specified in no more than one place, or that identical initialization must be specified in all declarations (to permit copying of declarations).

Requirement D6

Our comments on this requirement are contained in the paper "A Note on 'Pointers'" by Christopher Ernest (September 1, 1976), presented at the Cornell conference.

Requirement E1

The need to be able to define new data types is clear. However, at the risk of being thought unfashionable, we maintain that new data operations should not be definable in the language, except by means of procedures and functions. In other words, new infix operators are neither necessary nor desirable, for the following reasons:

- (1) H2 precludes the definition of new precedence levels, and we agree -- too many levels is confusing. But this means that a new infix operator must be at the same precedence level as an existing operator, which can also lead to distortion, and which means that new operators are clearly different from built-in ones. The problem goes away if new infix operators are not allowed; functional notation specifies precedence clearly.
- (2) If new symbols are introduced as operators, then very soon the APL problem of overpunching, etc., arises -- that is, the primitive alphabet becomes too large for ready comprehension. If words are used as new operators, perhaps enclosed in special brackets, then what is the advantage over function notation?

- (3) The requirement for component by component operations (R7) means that the most natural way of introducing new operators -- as new meanings for existing operators -- is often not possible. Matrix multiplication, for example, must be expressed by means of a new operator (as it is in APL). Again the advantage over functional notation is unclear.
- (4) Too many operators makes a language very unreadable, in our view and in Dijkstra's view also. He has strongly criticized APL for exactly this. More generally, he emphasizes the need to restrict, rather than to expand, the basic tools we work with, and we concur wholeheartedly. Addition of new operators to a language should not be easy; making it so invites abuse.

More philosophically, we would argue that a language is best extended through its literature (i.e., for a programming language, through its programs), in which existing basis elements are combined in new ways. This is certainly true of natural languages: New words are introduced very rarely, and new meanings for existing verbs (the operators of the language) almost as rarely.

We believe that the set of operators which deserve to be generic grows slowly. The basic set of operators should probably include more than the Tinman requires -- for example, matrix operations and complex arithmetic -- and subsets of the operators should be defined to be supported by different level compilers. Then before a new operator is introduced, the need should be clearly demonstrated, and if warranted, the compilers and the language documentation would be modified to support it. Such additions should happen rarely, and implementation through the compiler often leads to greater efficiency. If someone does not understand enough about the structure of a language to understand its compiler, should he in fact be able to modify the language?

These comments do not apply to new types or new data structures. These can be thought of as adjectives applied to a noun to restrict its scope, or as strings of nouns, embodying additional attributes. In other words, the language must be able to deal with new kinds of objects. The above comments do apply to the introduction of new ways of defining data structures, and here we and the Tinman seem to agree.

It is of course necessary to be able to do something with new types, and new data structures. The only way to introduce new scalar types is by enumeration, and all enumeration types share a common set of operations. The interesting cases are new structured types. The structure-defining mechanisms must include generic specification of the way in which individual components of a structure can be referenced and modified; for example, all array types have the same access conventions, whether their members are real, integer, or themselves structured. It is also possible to define generic rules for the meaning of relational operators with a new ordered type; if these are not sufficient, functions can be used. In some cases, reference to or modification of a variable should have side effects -- for both built-in and new types.

TINMAN
Requirement E1

8

The Tinman appears to partially share these views -- Boolean could be provided as an enumeration type, with sufficient definition of new operators. The fact that it is specified explicitly as a built-in type seems to indicate a realization that new operators should affect the base language.

Requirement E4

This requirement conflicts directly with requirement R7. Independently of that, we disagree with the requirement for reasons explained at length in the comments on requirement F1.

Requirement E7

We recommend that the following requirement be added: "It must be possible to compile a program so that there is no overhead caused by run time type checking." This is intended as a language requirement. Discriminated union implies run time type checking, supplied by either the compiler or the programmer (as in ALGOL 68). For a fully debugged program, the overhead of this must be avoidable. If the language insists on, for example, a tag field for a discriminated union, then the compiler cannot remove it even if run time checking is not wanted. There are many ways in which a program can determine the type in a given instance; a tag field should not be mandatory.

Requirement F2

This requirement, together with requirement E5, implies strongly that the Tinman intends a type and its operations to be defined in an isolated "cluster", with the objects of the type inaccessible outside the cluster except using the defined operations. This does not provide for new operations which have arguments of more than one type, and which must have access to the representation of the data objects. For example, if a new set type is defined for sets built at run time, an "add member" operation might need to access both a (normally inaccessible) component of an element and the control information for the set.

Better than a "cluster" for each type definition is a "cluster" for a collection of type definition. The Tinman should make this the requirement.

Requirement F6

while the meaning of the term compool is less restricted than it once was, it has remained true that a compool contains only things useful at compile time. Moreover, a library might contain a compool, but why must a compool be able to contain a library? These distinctions seem useful. We don't see the advantage of merging two distinct concepts into one.

Requirement 62

The Tinman's attitude toward the goto seems ambivalent. It is not eliminated entirely, and there is no restriction against using it to exit from an entire nest of loops (see G4). (Our reasoning is as follows: Tinman does not explicitly require any control structure to be a scope. The use of a goto is restricted to its most local scope. If control structures are scopes, then the goto is thereby restricted to the point of uselessness or, worse, to being used poorly only. Therefore, Tinman indeed did not intend for control structures to be scopes.) Yet there is a restriction against using a goto out of an allocation or procedure scope. As requirement G7 recognizes, such exits are clearly necessary; special mechanisms are therefore required. Exception handling parameters (requirement C7) are required as one such mechanism; no mechanism is identified, or even explicitly mentioned, for exit from allocation scopes.

Note that any implicit actions (e.g., release of storage, stack popping) that would be connected with a goto out of an allocation or procedure scope are necessary in any case for the special exit mechanisms. Less complex, but similar, mechanisms are needed for exit from a nest of loops. If the use of a goto is thought to be confusing or error prone in the first case, why not in the second also? That is, if special exit mechanisms are required, why not spell out exactly what is wanted where, and eliminate the goto entirely?

In fact, our recommendation would be to keep the goto, and allow it to be used in an almost unrestricted manner (a prohibition against using a goto to enter a control structure or an allocation scope is certainly reasonable), but provide enough other tools so that its use was rarely necessary.

Requirement 63

The use of the complementary clause of a control structure (*else after if then* or *otherwise after case*) with a null statement list occasionally makes a source program easier to read. This usually occurs when the set of statements following an *if then* or the partitioning being specified by a *case* is complicated, for example. In such cases the complementary clause has the visual effect of a closing bracket for the control structure and tends to convince the reader that he has not

missed anything. It seems a bit severe for the language to require the complementary clause, however. For example

IF X < 0 THEN X = X + 1;

would simply become cluttered by the inclusion of an else clause. Such questions should probably be left to individual style. (Even in tightly controlled developments programmers should be allowed some freedom.)

Requirement G4

We applaud the requirement that a terminating predicated be allowed to appear anywhere in the loop. As noted above, this also strengthens our goto argument.

Requirement G5

We concur wholeheartedly with the decision to make recursive procedures a requirement. We believe that the costs of recursion are justified by its contribution to clarity and conceptual simplicity for a user.

however, we do not agree with the restriction against nesting recursive procedures. We do agree that such nesting occurs rarely -- which means the display has only one entry normally anyway, if the compiler knows which procedures are recursive. The restriction therefore has no real run time advantages, and we would prefer a requirement calling for declaration of the fact that a procedure is recursive. In general, a compiler cannot develop precise information for itself (e.g., the fact that R calls P and P calls R does not necessarily mean that either is recursive). With or without the restriction against nesting, the declaration is required to avoid extra cost.

If the restriction against nesting of recursive procedures is nonetheless kept, then we note that it could be relaxed somewhat. It is only recursive procedures in recursive procedures which cause a multi-entry display. Other uses of stack storage (e.g., loc) may cause additional pointers, but the implementation choices are the same here with or without recursion. It is a bit tricky to avoid extra costs due to recursion when calling (non-recursive) formal parameter procedures, but it can be done.

Requirement G8

As the Tinman correctly remarks, synchronization of parallel processes can be done through exclusive access to data, required by G6. It can also be done by calling operating system or other synchronizing routines. Hence, the meaning of this requirement is unclear -- must additional synchronization tools be provided, or are the two mentioned enough? We would opt for the second alternative; additional tools should not be outlawed, but neither are they necessary, and deciding on a set compatible with various different environments is very hard. Note that the language BLISS, which supports asynchronous processing, provides no synchronizing primitives, presumably for these reasons.

The meaning of assigning priorities is also not clear. Typically, relative priorities change with time, to preclude endless waits for low priority processes, or to make it possible to meet deadlines. We certainly oppose building a comprehensive scheduling mechanism into the language, because different operating system disciplines would render much of it meaningless. Note that scheduling can be programmed in any way desired, by delaying processes to wait for access to shared data. Again, it is not clear why the additional requirement is necessary, and what it accomplishes. In a particular environment, or for particular uses (e.g., simulation), scheduling and specific synchronization mechanisms can of course be programmed, but requiring general ones seem untenable. The problems are illustrated by CS-4, which has nine scheduling procedures, four synchronizing ones, and six parameters describing priority! It is not clear that these are meaningful in all cases, or even that they are sufficient.

Treatment of hardware and other interrupts is a different question. Here it would be possible to provide tools in the language, as PL/I has done, for example. However, these add quite a bit of mechanism to the language. It should be possible to tell what statement caused the interrupt (where applicable). It must be possible to enable and disable interrupts, to specify the action to be taken on interrupt as a special kind of procedure or process, to cause control to continue where interrupted (and exact definition of this is extremely hard) or to exit from the interrupt procedure without resuming. Debugging of interrupt procedures presents special problems (in PL/I, SIGNAL and CONDITION can be used, but this is not quite the same as an actual interrupt). Moreover, control over interrupts has many of the undesirable traits of the goto; control is interrupted in a definitely "non-structured" way. Therefore, we question the requirement -- the added complexity may not be worth the gain.

In summary, we agree only with the part of the requirement calling for access to real time clocks, and for delays based on elapsed time or the occurrence of a specified time. The other facilities called for should certainly not be outlawed, but neither should they be required of the language.

Requirement H4

We suggest the addition of a requirement that every literal be self-identifying as to type. To prevent confusion, and make parsing easier, context should not be necessary to determine the type of a literal. The requirement does not preclude the use of immediate qualifiers, as in for example:

complex: (1.3,4.7)

Requirement 14

The requirement of conditional compilation is a good one, but we believe strongly that statements which are to be executed at compile time should be clearly identified as such. Otherwise, the flow analysis problems for the translator can be severe, and there is no reason to impose this extra burden.

Requirement J?

It is difficult to understand this requirement as a language requirement. Any optimization will change the effect of a program in some sense. Any so-called optimization which changes the semantics of a program is actually a bug in the compiler. If this requirement needs to be stated, it should be moved to section L or M.

Requirement J3

A useful feature would be to require the encapsulating statements to contain information specifying the hardware features for which the dependent code is being written. When a program written for one object machine is compiled for another, the compiler would be able to flag machine dependent sections which have not been modified. (Notice that including machine dependent text within compile-time conditional segments which depend on some compile-time variable set by the programmer merely helps generate an error-free source listing. It does not point out sections of the program which must be modified.) For assembly language segments the specifying information could be as simple as the symbolic name of the object machine. The identifiers for the symbolic names could be administered by the language controller, to ensure uniformity of implementations, and would not have to be treated as reserved words.

Requirement J4

As we suggested under requirement J3, the hardware features assumed should be specified in the encapsulating statements, but here the specifications could often be more general than simply the object machine. For example, the number of bits per word is a feature which could span a number of machines.

SECTION 3 - COMPARATIVE EVALUATIONS

This section is tabulated into eight subsections, each containing an assigned language which Computer Sciences Corporation evaluated against Department of Defense requirements. The languages are:

CMS-2	JOVIAL J3B
CORAL 66	JOVIAL J73
CS-4	PEARL
EUCLID	SPL/1

A COMPARISON OF
CMS-2
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language CMS-2 to the Tinman Language Requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CMS-2 is considered to be defined by:

Users Reference Manual for Compiler, Monitor System-2 (CMS-2) for Use with the AN/UYK-7 Computer M-5035, Vol. 1 and 2
FCDSSA - San Diego, Cal.
15 August 1975

CMS-2Y Programmer's Reference Manual (Preliminary Version)
M-5049
FCDSSA - San Diego, Cal.
1 October 1976

Tinman contains 78 language requirements. This report compares CMS-2 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CMS-2 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....P

The language permits the user to explicitly specify the type of each variable and component of a composite data structure. The entity and its type may be declared with a variable or field (component of a record) declaration statement. If the type is not specified in the declaration the entity is assigned the type specified by a mode declaration, or, if there is no mode declaration, a default type. In addition, an optional declaration (OPTIONS MODEVRBL) allows the user to use local variables without being required to declare them. The type assigned to them is the type specified by the mode declaration or default. In addition a table may have a major index which dynamically controls the number of actual items in the table. Specification of a major index is considered a variable with signed integer typing of 16 bits length.

Since the mode declaration and MODEVRBL option are error prone (even though the compiler "alaringly" flags all MODEVRBL data units) these options are seldom used. It is a matter of policy for users to declare all data units.

Removal of the MODEVRBL option so that all entities must be declared and requiring a major index to be pre-defined would be simple modifications to the existing implementation.

Type checking is performed at compile time. However, the full intent of TINMAN is not met. All numeric data units are considered to be the same type for assignment and expression evaluation and implicitly converted if necessary. (See comments under A2 and B8.) Data units may share the same storage (OVERLAY feature) creating a free union, but discrimination is not permitted. (See comments under A7.)

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....T

IntegerT
Floating PointT

Fixed Point	T
Boolean	F
Character String	T
Arrays	T
Records	, T

All of the above are built-in types in CMS-2. For type checking purposes all numeric data units are considered to be one type.

CMS-2 has a data type called a TABLE which consists of any number of items. An item is an arrangement of atomic data units, i.e., a record. A TABLE may be an array of items (records) or a single item (record) but a record may not contain an array or a record. To permit records to contain arrays or other records would require a major change to the language and its implementation.

In addition to the binary valued Boolean (true, false), the language contains a multi-valued Boolean or bit-string. Only the binary valued Boolean may be declared. But any data unit can be considered to be a bit-string and its bits may be accessed by the BIT modifier or manipulated by the Boolean operators AND, OR, COMP (complement) and XOR.

CMS-2 also includes an enumeration type, the status variable.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.F

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedF

The language does not permit a precision to be specified for floating point arithmetic or for individual floating point data units.

CMS-2 is a language that compiles for a single machine, the AN/UYK-7. Floating point data units correspond to the AN/UYK-7 doubleword format for floating point numbers. They always have a precision of 31 bits.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.P+

Treated as exact quantitiesT
Range and step size determined at compile timeP+
Scaling handled automaticallyT

Fixed point numbers are treated as exact quantities. The minimum number of bits needed to represent the number, and the number of fractional bits are specified by the user in the declaration. This determines the maximum absolute value the number can assume. The minimum absolute value cannot be specified. Scale factor management is handled by the compiler unless overridden by the scaling specifier appended by the user to arithmetic expressions.

The language has a DEBUG option which permits the user to specify a complete range for each data unit. Instructions are generated to provide dynamic range checking. Since the mechanism is available to accept a complete range specification it would be a relatively simple modification to permit a complete range to be specified in the data unit declaration statement. However, instruction generation for dynamic range checking should remain optional.

A5. Character sets will be treated as any other enumeration type.P-

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedP
(Conversion capability between sets is available)N/A

The implementation of the language uses the 64 character ASCII subset. No other character sets are permitted. The compiler assumes that all peripheral devices use the same character set.

To allow the user to specify the program literal and order of characters would require only a translation table for conversion from the internal representation and collating sequence used by the compiler to the character set and collating sequence specified by the user. Translation would only be necessary by the portion of the compiler that generates character constants, and for input/output with peripheral devices that use a character set different from that defined by the user.

If the character set defined by the user can have characters that are not in the 64 characters ASCII subset the character set recognizable by the compiler must be expanded.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryP
Subscripts only integers or from an enumeration typeP
Subscripts will be from a contiguous rangeT

The user must specify the number and size of each dimension. The subscript values can only be integers and must be contiguous values starting at zero. The maximum size of an array is fixed at compile time, except for simple tables (one-dimensional arrays) whose maximum size can be fixed at load time. In addition, the size of simple tables can be dynamic by declaring a major index, a variable that controls the upper bound on the table subscript. Declaration of a major index automatically defines the major index. Its type is a signed integer of 16 bits, and its scope is the same as the scope of the table.

Subscript values can be values of an enumeration type data unit only if the enumeration type data unit is explicitly converted to an integer by use of the BIT operator.

A lower bound on the subscript range other than zero or one would require a change to the array declaration syntax and a corresponding change in the symbol table.

Extension of the major index feature to all arrays would be a modest change to the compiler.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.P

Alternative structures for records are possibleI
Discrimination condition may be any Boolean expressionF

Records (items of a table) may have alternative structures either by user packing of the fields so that they share the same storage, or by use of the field overlay feature for compiler packed fields. The structures are fixed at compile time but cannot be discriminated at run time.

The scope of modifications necessary to implement run time discrimination for alternative structures is dependent on the syntax and semantics to be used.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.T

Automatically defined for any type (except...)T
Available for individual componentsT
(Assignment and reference via functions)F

Assignment and reference is defined for all data types including arrays and records and for their components. In addition, word components of arrays and records can also be assigned and referenced.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.P+

Any two atomic data units of the same type (all numeric data units are considered to be the same "type") may be compared for equivalence using the identity predicates EQ (equals) or NOT (not equals). Arrays may not be compared. Components of arrays (fields) may be compared if they are of the same type.

The modification of the existing implementation to permit comparison of conformable arrays would be minor.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P+

Built-in for all numeric and enumeration typesT
Ordering can be inhibited when desiredF

All six relational operations (EQ, NOT, GT, GTEQ, LT, LTEQ) are built-in for all numeric and enumeration types. It is not possible to inhibit the ordering of the values for an enumeration type.

An extra piece of syntax would be required to indicate an unordered enumeration type. The modification necessary would be trivial.

B4. The built-in arithmetic operations will include:
addition, subtraction, multiplication, division (with a real
result), exponentiation, integer division (with integer or
fixed point arguments and remainder), and negation.T

AdditionT
SubtractionT
MultiplicationT
Division with real resultT
ExponentiationT
Integer and fixed point division with remainderT
NegationT

Arithmetic operations use familiar notation: addition, +;
subtraction, -; multiplication, *; division, /; exponentiation, **.
Negation uses the unary operator -. A division with real (floating
point) result occurs if either of the operands is real. The remainder
of any fixed point division may be obtained by using the remainder
operator (SAVING <data unit>).

B5. Arithmetic and assignment operations on data which are
within the range specifications of the program will never
truncate the most significant digits of a numeric quantity.
Truncation and rounding will always be on the least
significant digits and will never be implicit for integers
and fixed point numbers. Implicit rounding beyond the
specified precision will be allowed for floating point
numbers.P

Never from the left for data within rangeT
Never on the right for integer and fixed pointF
Implicit floating point rounding beyond precision allowedP
(Run time checks can be avoided)T

?
For arithmetic and assignment operations, CMS-2 never truncates the
most significant digits if the data are within range. Implicit
truncation is performed on the right and a warning message is issued.
For floating point data units the user can specify in the data
declaration whether implicit rounding is to be performed or no rounding
is to be performed.

Run-time checks for truncation could be added at modest cost.
However, run-time checks are costly to the user.

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P+

Short-circuit andT
Short-circuit orT
NotT
XorP

CMS-2 has two sets of Boolean operations: Boolean connectors (i.e., logical connectors on Boolean data units or expressions as in SET <Boolean data unit> TO <Boolean data unit> AND <Boolean data unit> \$); and Boolean operations on bit strings (as in SET <integer> TO <integer> AND <integer> \$).

For Boolean data units the language provides AND, OR and COMP (complement). The implementation of AND and OR is short circuit.

For bit strings the language provides AND, OR, COMP and XOR.

Implementation of the logical operator XOR would require a moderate addition to the logical code generation sequences.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.P

Scalar operations on arraysP-
Assignment between records and arrays of conformable typeT

The only scalar operation permitted on arrays is assignment. For example, SET TABARY TO 5 \$ where TABARY is an array. Each word of TABARY is assigned the value 5. All scalar assignments to tables (arrays) and to items (records) of tables are word assignments.

Assignment between tables and between items of tables is permitted without regard to the types or logical arrangement of the components of the tables or items. The tables or items do not need to be conformable, nor of the same length. Transfer is accomplished on a word basis, the length of the shorter determines the number of words transferred.

To require type checking and conformability checking on array and record transfers and to allow logical assignment between non-conformable arrays would require a moderate modification of the current compiler.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

....P

No implicit conversionsF
Explicit between integer, fixed point, and floating pointF
Explicit between fixed point scale factorsT
(Explicit between integer and Boolean)P
(Explicit between integer and enumerated types)P
(Explicit between different enumerated types)F

In CMS-2 all numeric data units are considered to be the same type. During expression evaluation and assignment implicit conversions are performed on integer, fixed point and floating point data units. Explicit conversions between integer, fixed point, and floating point data units are not required nor permitted. However, scale factors can be explicitly specified on fixed point data units and fixed point data unit expressions.

An explicit conversion can be performed on a Boolean, a character type or an enumeration type to an integer by the use of the BIT operator. An integer (or any other numeric type) may not be converted to a Boolean or an enumeration type. Any numeric data unit or any data unit converted to a bit string by the BIT operator, may be converted to a character type by use of the CHAR operator (if the number of bits is a multiple of eight).

Replacement of the implicit conversions on numeric data units by explicit conversions would require a moderate change to the existing compiler.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is

truncated.

....P+

Implicit conversion between rangesT
Exception condition on integer and fixed point truncationP

Explicit conversion operations are not required between two numerical data units that are out of range. If truncation occurs the compiler issues a warning message.

A run-time exception condition for truncation could be added at modest cost.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

....P+

Sending and receiving of dataT
Sending and receiving of control informationT
Dynamic device assignmentF
User exception condition controlP
Installation independenceT
(Data formatting capability)T
(Reading and writing of bit strings)T

CMS-2 has a rudimentary high level I/O capability. With it the user can transfer data between his program and named files on tape or disk, and standard I/O devices (card reader, printer, punch, and operator's console), in a formatted or unformatted form. The I/O operations require the use of library run-time routines and the CMS-2 monitor.

The CMS-2 high level I/O capability provides the equivalent of what one would need in a typical batch environment, and is not sufficient for tactical data systems. The I/O functions in embedded executives in most tactical data system applications are usually written in direct (embedded assembly language) code.

Expansion of the high level I/O capability to provide the functions necessary for tactical data system applications would be a major effort.

B11. The language will provide operations on data types defined as power sets of enumeration types (see F6). These operations will include union, intersection, difference, complement, and an element predicate.

....F

Union	F
Intersection	F
Difference	F
Complement	F
Membership predicate	F
(Set inclusion)	F

Although an enumeration type (status variable) is a permissible type the powerset of an enumeration type cannot be defined, nor can any of the set operations.

The inclusion of powersets and their operations would have no impact on existing features in the language and could be added at modest cost.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.T

Side effects must occur in left-to-right orderT
(Embedded assignments)F

Those arguments of an expression that can cause side effects are evaluated left-to-right.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.Pt

Few precedence levelsT
No user-defined precedence levelsT
Operands of an operation are obviousPt

CMS-2 has six precedence levels. Within the same precedence level evaluation is from left-to-right unless the operation to be performed first is specified by parentheses or a unary minus is encountered before or after an exponentiation operation, in which case the operations are performed from right-to-left. The left-to-right evaluation defines $X/Y/Z$ as equivalent to $(X/Y)/Z$. The only exception to these rules is the expression evaluation for substitution declarations. See the comments on requirement H10.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.Pt

CMS-2 permits expressions anywhere both constants and variables are allowed. The form of an expression for substitution declarations is different from expressions in other contexts. See the comments on requirement H10.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

....T

This requirement is fully met.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.

....T

Parameter rules consistent in all contextsT
No special operations applicable only to parametersT

There is a consistent set of rules for each class of parameters.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

....P

Actual and formal parameters will agree in typeP+
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedP

The type rules for actual/formal parameters are the same as for assignment: Actual and formal parameters must agree in type except that all numeric data units are considered to be the same type.

The size and subscript range of parameter arrays cannot be explicitly passed. However, a formal or actual procedure input (or output) parameter can be a table with a major index. The major index on a simple table, i.e., a one dimensional array, can be passed explicitly as a parameter.

See the comments on A6.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.?

Act as constants (call by value plus)	F
Act as variables (call by reference)	P-
Exception control	T
Procedure parameters	F
(Act as variables, but call by value)	T
(Act as variables, result parameter)	T

A procedure in CMS-2 may have input parameters, output parameters and abnormal exit parameters. Abnormal exit parameters are label names on the calling side and condition names on the called side. The label names need not be within the scope of the calling procedure but must be within the scope of the system-procedure containing the calling procedure.

Input and output parameters are called by value (except for input of indirect tables). Assignment to the formal parameter is permitted within called procedure. Assignment to formal input parameters does not affect the actual input parameters.

Indirect tables are the one exception to call by value. The address of any simple table (by use of the CORAD functional modifier) may be passed as an actual parameter to an indirect table formal input parameter, and hence is a call by value. Assignment to the formal parameter effects the actual parameter.

A redefinition of formal parameters would result in a major impact on the existing implementation.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.F

Above properties optional	F
Above properties are fixed at run time	F

CMS-2 does not permit the writing of generic procedures.

Implementation of this requirement would require major modifications to the existing parameter mechanism.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.F

Variable number of arguments possibleF
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeF

CMS-2 does not permit a variable number of arguments.

Implementation of this requirement would require major modifications to the existing parameter mechanism.

D1. The user will have the ability to associate constant values of any type with identifiers.T

By use of the EQUALS and MEANS substitution features an identifier can be associated with any constant.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesT
Consistent interpretation in program and dataU

Literals are provided for all atomic data types. The documentation is not clear as to the consistency of numeric constants in both programs and input or output data.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.T

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesT

The initial value for any numeric, character, Boolean or enumeration variable or field (component of a record) can be specified as part of its declaration or can be specified separately with a data declaration statement. If not specified there is no default value. All data units are allocated at the time of their declaration. Initialization of arrays and records must be accomplished by declaring initial values for their components.

D4. The source language will require its users to specify .

individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....P

Numeric variable range specification mandatoryP
Fixed point variable step size specification mandatoryT
Range and step size specifications do not define a new typeT

For fixed point numeric variables and fields (components of a record) the maximum absolute value and the step size must be specified by declaring the maximum number of bits required to represent the numeric and the position of the binary point. The minimum absolute value cannot be specified and is assumed to be zero. The range for floating point numerics cannot be specified.

The language has a DEBUG option which permits the user to specify a complete range for each data unit. See the comments for requirement A4.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

....P

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataP

The values which can be associated with a variable or a record component (field) can be any atomic data type. A component of a record cannot be an array or another record. A component of an array can only be a record or an atomic data type. (See the comments on requirement A2.)

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....P-

Recursive and network structures providedF
Handles variable-value and structure-component connectionsF
Pointer property is an attribute of a typed variableF
Pointer property not for constants, affects only assignmentF
Pointer property mandatory for dynamic allocationF
Allocation scope never wider than access scopeF
(Either the value or the pointer is modifiable)F
(Pointer mechanism handles procedures and parameters)F
(Remap and replace assignment have different syntaxes)F
(Built-in dynamic variable creation)F
(Variable equivalence classes are declarable)F

There is a very limited pointer mechanism available by use of the intrinsic CORAD function. The address of any data unit can be referenced and assigned to any numeric variable or field (component of a record). However there is no mechanism for indicating that the value of a variable is an address. The CORAD function is used primarily for indirect tables.

The full impact of the implementation of a pointer mechanism cannot be ascertained without a full description of the syntax and semantics. However, any implementation of a pointer mechanism would be a major modification to the compiler.

E1. The user of the language will be able to define new data types and operations within programs.P

New operations are definable only by the use of functions. Since almost all languages have functions it is assumed that the use of functions as new operations definitions is not the intent of TINMAN.

New types are definable by only enumeration (status variables) or by records (items of a table). See the comments on requirement A2.

Adding the capability to define new operations to the language beyond the use of functions would require major modifications to the language and its implementation.

E2. The "use" of defined types will be indistinguishable from built-in types.F

The only definable types, enumeration and records, are also built-in. Although, as stated, the requirement could be considered to be literally met, the intent of Tinman is not.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.P

Local variables need not be declared if OPTIONS MODEVRBL has been declared, and the type of a variable (or component of a record) may be omitted. See the comments on requirement A1.

System and local indexes, when declared, are defaulted to signed integer types with a length of 16 bits. Removing system or local indexes would be a trivial modification. Allowing system and local indexes to be declared any type would be a modest change to the language and its implementation.

E4. The user will be able, within the source language, to extend existing operators to new data types.P-

The only defined types permissible are the enumeration type and the record. Assignment can be extended to both of these types. In addition, all of the comparison operations can be applied to enumeration types.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.F

ConstructionF
SelectionF
PredicatesF
Type conversionsF
Operations and data can be defined togetherF

There are no definable operations in CMS-2.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the powerset of an enumeration type. These definitions will be processed entirely at compile time.P

EnumerationT
Cartesian products (records)T
Discriminated unionF
Powerset of an enumeration typeF

The only ways to define new types is by enumeration (status variables) and by records (items of a table). Implementation of discriminated union and powerset of an enumeration type would require a major modification to the language and the compiler.

E7. Type definitions by free union (i.e., union of

non-disjoint types) and subsetting are not desired.

....P-

Free union is permitted using the overlay features. Subsetting is only permissible on arrays (sub-tables).

The overlay capability can be easily removed.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

....F

InitializationF
FinalizationF
Allocation actionsF
Deallocation actionsF

CMS-2 does not provide any encapsulated means of defining initialization, finalization, allocation or deallocation actions for defined types.

Any other means of defining actions would require a major modification to the language and the compiler.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

....P-

The scope of allocation, or lifetime, of all data structures (except local indexes) is the entire life of the program. The scope of access for data structures defined in system-data-designs is the entire program, i.e., the data structures are common to all procedures. The scope of access for data structures defined within local-data-designs is the system-procedure which contains the local-data-design, i.e., the data structures are common to all procedures within the system-procedure.

A local index is defined at the beginning of a procedure and exists throughout the procedure. Its scope of allocation and scope of access are identical.

What is missing in CMS-2 is the capability of limiting the scope of access for any data structure to a procedure or to any lower level. The changes to the compiler necessary to incorporate this capability would be a moderate modification to the symbol table and the symbol table accessing mechanisms.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

....P-

Allowable operations can be limitedF
Access can be limited where usedP-
External declarations need not all have the same scopeF
Naming conflicts can be avoided (renaming)F

Access to data structures or program structures is limited by the location of the data or program structure definition. Data structures are either global to the entire program, local to the system procedure or local to a procedure (local indexes). See comments under requirement F1. Access rules cannot be specified either where the structure is defined or where it is used, except that it is possible to define a structure (either procedure or data) as being global to the entire program, by use of the EXTDEF modifier, even though its location would make it local to the system-procedure.

The modifications necessary to define access where the structure is called cannot be assessed at this time.

F3. The scope of identifiers will be wholly determined at compile time.

....P+

The scope of identifiers is wholly determined at compile time. An identifier that is global cannot be duplicated at a local level. However, an identifier that is local to one system-procedure and an identifier that is local to another system-procedure can have the same name.

Lexically embedded access scopes would only be meaningful if the language provided for data definitions at a level lower than system procedures. At present the only data definitions permissible within a procedure is a local-index. See the comments under F1.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

....T

Applications oriented data, procedures, and functions can be stored in common libraries that are easily accessible in the language.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

....P

Program component libraries accessible at compile timeT
Libraries can contain foreign language routinesU
Interface requirements checkable at compile timeF

Program components and data in either source or object form or compools can be selected from compile time accessible libraries. Provided that the object output from foreign language compilers is compatible with the CMS-2 UYK-7 loader there does not appear to be any reason why a library cannot contain a program component that had been originally written in a foreign language.

F6. Libraries and Com pools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized com pools or libraries any user specified subset of which is immediately accessible from a given program.Pt

Libraries and com pools will be indistinguishablePt
Immediately accessible sublibraries at any levelT

CMS-2 libraries are collections of program elements (system-procedures and system-data-designs). System-procedures may be in object or source format. System-data-designs may be in object, source, or partially compiled (compool) format. The language permits the selection of any program element that is in source or compool format. (The librarian and loader may select any element in object format.) However, if the element is in compool format the user must specify that it is a compool that is being selected.

Changing the compiler so that the user would not have to specify the kind of program element would be a minor modification. From the user's point of view the compool would be indistinguishable from any other program element.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.P-

CMS-2 provides a limited high level I/O capability and interface with hardware. See the comments on requirement B10.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

....P

Sequential execution	T
Conditional execution	T
Iteration	T
Recursion	F
(Pseudo) parallel processing	F
Exception handling	F
Asynchronous interrupt handling	F
Control structures from a small set of simple primitives	"-

Sequential execution: Unless a statement involves a branch the next statement to be executed is the statement immediately following. In addition, statements may be connected with a THEN, as in: IF <condition> THEN <statement> THEN <statement> ... ELSE... or <statement> THEN <statement> ... \$.

Conditional execution is provided for by the IF <condition> THEN ... ELSE structure, the case structure (FOR <expression>), and switches.

Iteration is provided for by the VARY block structure.

Exception handling: abnormal exit parameters and overflow condition.

No syntax extension capabilities exist in CMS-2 to build new control structures. More powerful, and/or special purpose control structures have been provided, traditionally, by the addition of new primitives to the language, e.g., FIND is a table searching control structure combining iteration and a conditional. A high level macro capability could be implemented for a moderate effort.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

....P

The language provides a GOTO operation. However, in CMS-2Y the GOTO label can be to any label within the system-procedure. Later versions of CMS-2 (such as CMS-2M) restrict the GOTO to procedure scope. In addition, the language permits switches.

Imposing that GOTO labels be restricted to procedure scope (or to any other scope rules) and eliminating switches would require only a minor modification to the language.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.Pt

Based on Boolean expressionT
Based on type from discriminated unionF
Based on computed choice among labeled alternativesT
All alternative must be accounted forF
Simple mechanisms will be supplied for common casesP

Conditional control is provided for by the IF-THEN-ELSE, the case statement (FOR) and switches. The IF-THEN-ELSE is a common case for selection of the alternatives based on the value of a Boolean expression. The case statement and switches provide for a computed choice among labelled alternatives. For all of the conditional control mechanisms the language does not require that all alternatives be accounted for (e.g., an ELSE is not required to be specified after the IF THEN). Discriminated union is not permitted by the language.

The case statement was added to the language to provide the same capabilities as the switch mechanism in a more structured form and to localize the scope to the structure. The switch mechanism can be removed with relatively little impact.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).Pt

Termination can occur anywhere in the loopPt
Multiple terminating predicates are possibleT
Entry permitted only at the loop headF
Simple cases are clear and efficientT

Control variable is local to the loopP-
Control value is efficiently available after terminationP

Iterative control is provided for by the VARY statement. Termination can be after a fixed number of iterations, at the beginning of an iteration upon failure of a test (WHILE), or at the end of an iteration upon failure of a test (UNTIL). The only termination of the loop permitted within the loop is by use of a GOTO outside the loop. Termination of any iteration within the loop and commencement of the next iteration can be accomplished by a RESUME statement. In addition, the RFSUME statement can be used outside of the loop if the loop was terminated by a GOTO outside the loop.

Entry is permitted into the loop at any labelled statement.

If a data unit is specified as the loop control variable the data unit is global to the loop structure and hence has a value after loop termination that is available. If no data unit is specified as the loop control variable then the number of iterations is not available at loop termination as no loop control variable is used.

If the loop is terminated by a GOTO the value of the loop control variable (if there is one) is available unless it is explicitly modified. Modification of the language to make loop control variables and statement names within the loop local to the loop (so that entry is permitted only at loop head) would require a moderate modification to the scope rules and their implementation mechanisms. See the comments under F1.

65. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.F

No recursive procedures within recursive proceduresN/A
(Maximum depth of recursion can be specified)N/A
(Recursiveness must be specified)N/A

The language does not permit recursion.

Procedure linking in present implementations of the language is accomplished without the use of a procedure linking mechanism. The return address for the calling procedure is saved in the called procedure. Since there is no data that is local to a procedure (except for local indexes) there is no data allocation or management that must be performed with each procedure call. These features permit CMS-2

programs to execute without requiring the overhead of a monitor or operating system. If the language were modified to permit procedures to be recursive, an external procedure linking mechanism with a stack for the returns and for data that must be preserved (such as local indexes and loop control variables) would be required. Without changing the scope rules to comply with requirements of Section F, the external procedure linking mechanisms to provide recursion would be a minor addition to the present implementation. However, recursion without variables local to the procedure may not be of much value. If the scope rules are changed a more extensive redesign effort would be required.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.F

Able to create and terminate parallel processesF
Process can gain exclusive use of resourcesF
No parallel routines within recursive routinesF
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF
(Access rules are enforced)F

CMS-2 provides no capabilities for parallel processing.

Extensive modifications to the language would be required to include parallel processing capabilities.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.P-

Program can get control for any exceptionF
Parameters can be passedP
Can get out of any level of a nest of controlF
Can handle the exception at any level of controlF

The only exception handling capabilities provided for in the language are the abnormal exit parameter (see G7), the overflow detection, the parity check, and the range check on an item index into a table.

The evaluation of an arithmetic expression in an assignment operation can be tested for overflow and control transferred to a label. (SET <data unit> TO <expression> OVFLOW <label> \$).

The Boolean condition in a conditional control statement can be a test for odd or even parity on any data unit (contained in one word). (IF <data unit> EVENP (or ODDP) THEN ...)

The Boolean condition in a conditional control statement can be a test that an item index into a table is within range. (IF <table index> VALID (or INVALID) THEN ...).

The extensions to the compiler necessary to provide a general purpose exception handling control structure would be moderate.

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

.....F

Priority specificationF
Synchronization via wait/enable operationsF
Wait for end of real time intervalF
Wait for end of simulated time intervalF
Wait for hardware interruptF
(Can enable and disable interrupts)F

None of these control features are provided for in the language.

A moderate extension to the language would be required to incorporate these features.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....P

Free format with statement terminatorT
Mnemonic identifiers possibleT
Based on conventional formsP
Simple grammarP
No special case notationsP
No abbreviations of identifiers or keywordsP+
Unambiguous grammarP

The language is free format. All statements must terminate with a 's'. Identifiers can be from one to eight alphanumeric characters.

With some exceptions the language is based on conventional forms: Assignment is SET <data unit> TO <expression> \$, the case statement uses the keyword FOR.

The grammar is simple for the commonly used features. However, there are many special purpose and seldom used features each with its own syntax. Complicated expressions involving Boolean operators and relational expressions can be unclear. There are some features that are context dependent. For example, arithmetic expressions in EQUALS substitution declarations are evaluated using a different set of precedence rules than those used for the evaluation of other arithmetic expressions.

The language could be changed easily to remove the seldom used features and special uses and to resolve the ambiguities. The implementation of these changes would be a modest effort.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.T

The user cannot modify the syntax of the language. However, injudicious use of the MEANS text substitution mechanism may have the appearance of syntax modification. For example, if an entire statement is substituted, the substitution name would have the appearance of a procedure call. Since a substitution name is not easily distinguishable

from a procedure name the use of MEANS text substitution can add to the visual ambiguity of the language.

The MEANS text substitution mechanism can be easily modified so that the substitution name can be easily distinguished from any other name by including a special signifying character in substitution names.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.T

CMS-2 uses the 64 character ASCII subset.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.T

Break character existsF
(Literals are self-identifying as to type)P
(Bit-string literals for any type)F

(1) Identifiers are formed by one to eight alphanumeric characters. The first character must be alphabetic. No break character is permitted.

(2) a. Character string literals are formed by enclosing the string in parentheses and preceding the literal by an 'H', e.g., H(ABC). Separate quoting of each line of a long literal is not required: H(THIS IS A LONG LITERAL STRING). Any break character could be used since the literal is enclosed. It would be interpreted as part of the character string literal.

b. Numeric literals may be octal or decimal. In the default compiler mode octal numbers must be enclosed in parentheses and preceded by the letter O. All numbers not enclosed in parentheses are assumed to be decimal. The compiler mode may be declared to be octal, in which case all decimal numbers must be enclosed in parentheses and preceded by a D (or may be unenclosed and suffixed by a D). All other numbers are assumed to be octal. All numbers may contain an optional radix point and may be expressed in scientific ('E') notation. No break character is permitted.

A break character of space, underline, or tilde could be used with any of the numeric literals. Only a minor modification to existing implementations would be required.

A break character of space could not be used with identifiers without introducing syntactic ambiguities into the language. Any break character to be used with identifiers would not be of much value unless identifiers could be formed from some number of characters significantly larger than eight. Adding this feature to the language would require a restructuring of the symbol table in existing compilers, thereby reducing the number of symbols available to the user.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

....F

The language permits lexemes to continue across lines.

This capability could be removed with a relatively minor modification to existing implementations. However, a break character must be provided for long character string literals.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

....P

The language has 179 primitives (or key words). A primitive cannot be assigned as an identifier (i.e., the primitive is reserved) if ambiguities would be created in the user's program. There are 29 primitives that cannot create ambiguities and, therefore, are not reserved.

Making all primitives reserved would require a simple modification to the existing implementation.

The large number of primitives in the language is indicative of the variety of syntax available. The number of primitives could be reduced by removing from the language those control mechanisms that are redundant or seldom used.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

....P

Uniform comment convention	F
Look different from code	U
Bracketed by one or two characters	P
Can contain any characters	T
Can appear anywhere reasonable	T
Terminated by the end of the line	F
Compatible with automatic reformatting	T

The language has three comment conventions:

(1) Comment statement: COMMENT <string> \$. The comment statement cannot appear within any other statement.

(2) Bracket notes: Used to document one of the five bracket declarations (system declaration, system data declaration, system procedure declaration, head declaration, and local data declaration) <declaration><string> \$.

(3) Notes: strings delimited by pairs of consecutive single primes. They may appear in any statement wherever it is permissible to use a space. ''<string>''.

Since the notes convention could accomplish the functions of the other two comment conventions it is not necessary to have more than one. The other two conventions could be removed easily from the language with no impact on existing implementations.

H8. The language will not permit unmatched parentheses of any kind.

....T

The language does not permit unmatched parentheses.

H9. There will be a uniform referent notation.

....Pt

Almost all function calls and array element references use the same notation. The only exceptions are the two intrinsic functions BIT and CHAR. It can be argued that BIT and CHAR are not functions in the normal sense that the term is used, but rather operators or modifiers.

Subsequent versions of CMS-2 have adopted the uniform function notation for BIT and CHAR. This has required only a minor modification to the language and has had a reverse impact on the implementation. That is, implementation for subsequent versions required less code since one fewer syntactic form was necessary.

H10. No language defined symbols appearing in the same context will have essentially different meanings.P

There are several areas in the language where the meaning of a symbol is context dependent:

(1) AND, OR and COMP are used both as logical connectors between Boolean expressions and as bit string operators (e.g., X AND Y results in the bit by bit logical AND of X and Y). This dual use has created problems of ambiguity

(2) THEN is used both as a statement connector (<statement> THEN <statement>) and in the conditional control structure (IF <expression> THEN ... ELSE).

(3) The symbol period is used both as a label terminator and as a radix point. In addition, two periods is used for the scaling specifier and three periods is used to indicate a series of consecutive items in an INPUT or OUTPUT statement.

(4) There are three types of EQUALS declarations: numeric constant, the difference between the allocations of two addressable names, and the allocation of an addressable name. The type of EQUALS is dependent on the evaluation of the expression in the declaration. In addition, the expression is evaluated left to right with no hierarchy of operators. Numeric expressions in other contexts are evaluated hierarchically.

Elimination of any of these multiple purpose syntactic forms would require the introduction of new syntactic forms. In each case there would be no impact on other requirements and only minor modifications to the present implementations would be necessary.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.T

There are no implementation defaults that affect program logic.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.P

Defaults specified for don't care casesT
Programmer can override the defaultsP

The language includes defaults that affect only object representation. These include:

(1) Data representation: table packing - the user may define the internal position of each field or he may let the compiler determine the most optimal internal arrangement of the fields according to one of three user specified packing modes.

(2) Function/subroutine calls: intrinsic functions may be either open or closed dependent on the input parameters. For example, the BIT modifier is open if the bit string specified does not cross a word boundary; it is closed otherwise.

(3) Reentrant vs nonreentrant: the user may indicate that the code is to be reentrant. If not specified, the code is nonreentrant. Data for reentrant code may be specified so that multiple copies are generated.

Function/subroutine calls: to provide the additional capability for the user to be able to specify whether the function is open or closed would require major code additions to existing implementations.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the

object machine configuration.

....P-

The only compile time capability available to the user is the CSWITCH. CSWITCH brackets allow sections of programs to be selectively compiled. The CSWITCH is set outside the program but is not available for interrogation within the program by the user. Other compile time information such as the presence of the operating system, are available only to the compiler and not to the user.

Inclusion of other compile time variables would require a major extension to the compiler.

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

....P-

Conditional compilation is provided for only with the CSWITCH. See the comments under requirement I3.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

....F

The base of the language is the entire language and it is not small. There are no extension features for defining new capabilities in terms of the base.

The base of the language could be reduced to provide a relatively low level general purpose capability merely by removing those seldom used and special case features.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

....Pt

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels and the number of nested block levels are part of the language definition. The number of identifiers in a program however, is not part of the language definition and is not fixed by the translator but is limited by the amount of core storage available on the AN/UYK-7 system that hosts the compiler.

Changing the limit on the number of identifiers in a program to a language defined limit would require a simple modification to the existing translator.

The limits on array dimensions, the number of nested parentheses levels and the number of nested block levels are set so high that no program should encounter the limits. The length of identifiers, however, could be considered by many to be arbitrarily short. See the comments on requirement H4 for the scope and impact of modifications of this limit.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....T

Restrictions such as amount of run time storage, access to specialized peripheral equipment, etc. are not part of the language definition.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.U

No efficiency cost for unused featuresU
Efficient code can be produced for all featuresU

The current implementation of the language attempts to generate efficient code for the widest variety of programs: Sophisticated optimization techniques are used; for common special uses the compiler generates special sequences of code rather than standard sequences for all cases; only the run time support routines required by the program become part of the program. However, certain trade-offs exist within the compiler and it is not possible to ascertain that the most efficient code is generated for all programs. For example, certain features always use a run time support routine. There are probably some infrequently occurring special cases of these features for which generation of in-line code would be more efficient.

J2. Any optimizations performed by the translator will not change the effect of the program.T

Optimization by the compiler does not change the effect.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.T

Machine language insertions are permitted in CMS-2 programs in data designs and procedures provided the insertions are bracketed with the statements DIRECT \$ and CMS-2 \$. The machine language is equivalent to AN/UYK-7 assembly language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be

able to specify the time/space trade-off to the translator.
If not specified, the object representation will be optimal
as determined by the translator.

....Pt

Encapsulated specification of representation possibleP
Space/time tradeoff can be specifiedPt

The language provides the user with the capability of specifying the object representation of the items of tables (records or arrays). The user can either pack the fields himself by defining the size and arrangement of the fields in the table or he can indicate how the compiler is to pack the fields by specifying a packing descriptor (NONE, MEDIUM or DENSE) with the table declaration. The packing descriptor NONE signifies that the compiler is to allocate the maximum space (and, therefore, least execution is required). DENSE signifies the least space (and, hence, most execution). In addition to the packing descriptor the user can further control the arrangement of fields within user packed tables by using the field overlay capability (free union).

With either user packing or compiler packing the user does not have complete control over the arrangement of fields within the table. A field may not start in the middle of one word and end in the middle of another word, and multi-word fields must end on a word boundary.

For user packed tables the starting word and starting bit of each field is not encapsulated but is additional information appended to the field description.

All of the fields within a table must be either user packed or compiler packed. User packing and compiler packing cannot be mixed within a table.

Expansion of the user's control over object representation of items of tables by allowing mixed user packing and compiler packing, or adding new packing descriptors, etc. would require, in addition to syntax revisions, a modification of the packing algorithms. Giving the user complete control over the positioning of fields over word boundaries would require new coding sequences and/or additional run time routines.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

....F

Open/closed properties can be specifiedF
Open and closed versions have the same semanticsF

All user defined procedures and functions are closed.

The current implementation of the translator could be retrofitted to include the capability of open user defined procedures/functions without requiring a major redesign. However, additional storage in the symbol table would be required to save the open procedure/function for later expansion, thereby reducing the number of symbols available to the user.

Extraneous Features

We recommend that the following features of CMS-2, not required by the Tinman, be kept:

- * Character string type (assuming that the Tinman requires only a character type, not a character string type) and the concatenation operator, CAT.
- * Result parameters (OUTPUT parameters) for procedures.
- * Control of the internal structure of tables (arrays) through (1) specifying the internal arrangement of the records of a simple table (one-dimensional array) as either horizontal or vertical and (2) specifying the compiler packing algorithm to be used.
- * The ability to declare system procedures (modules) to be reentrant.

The following features provide a useful capability to the CMS-2 user community. They should be either kept intact or replaced by some equivalent feature:

- * Bit string type. A datum of any of the atomic types can be converted to a bit string by use of the BIT functional modifier.
- * The debug option. This capability has proved useful and could even be expanded.
- * The MEANS statement (temporary source substitution). This could be included in a more general macro capability.
- * The explicit conversion operators BIT and CHAR, which convert data to bit string type and character string type respectively, regardless of their declared type.
- * The scaling specifier for intermediate computations in fixed-point expressions.
- * Multiple receptacle assignment statements. The most common use for this feature is initialization at what would be the beginning of a scope in a more modern language.
- * Procedure switches (item and index). Although the same effect can be obtained through a combination of a FOR (case) statement and procedure calls, this syntax enables the compiler to generate code which is

efficient in both space and time, particularly if parameter passage is involved. NTDS takes advantage of this.

The following features are highly machine dependent, but they have been found useful by CMS-2 users. They should be retained because of this user interest, but they should be required to be bracketed (encapsulated) by statements which mark them as being machine dependent (requirement J3):

- * Designation that a procedure parameter is to be passed in a specific machine register.
- * Designation that a datum is to reside permanently in a machine register within its scope of definition (system and local indexes). This might even be expanded to allow various data types to reside in these registers; currently only one range of integer type is permitted.
- * The shift operators, which duplicate the machine shift instructions.
- * The stop operators, which duplicate the machine STOP instructions. A variant of this capability also occurs in the RETURN statement.
- * The COUNT function, which counts the number of 1-bits in a data unit.

The following features of CMS-2, not required by the Tinman, should be deleted:

- * The EXCHANGE statement (permanent source substitution). This is actually a function of a source file editor and should not be in the language.
- * OVERLAY, a form of free union.
- * Sub-tables (declared subdivisions of tables). This has most of the bad properties of free union with very few offsetting virtues.
- * The FIND statement (table search). This is simply a replacement for a combination of a VARY and an IF statement.
- * Statement switches (index and item). These are elaborations of the GOTO statements.
- * The ability to specify that a floating point datum is to be rounded. This is too hardware dependent.

- * The PACK statement. This is little used and is almost useless because its inverse (UNPACK) is not in the language.
- * The SWAP statement. This is too little used to be worthwhile.
- * Table word referencing, which permits addressing a machine word within a table (array). This is too hardware dependent, thereby seriously inhibiting portability, and is a means of defeating type-checking as well.

Summary

CMS-2 is an old language. The original design of the first version of the language was influenced by its predecessor (CS-1), the language concepts prevalent in the mid-sixties, and its first target machine (the AN/USQ-20). Changes to the language through the years were made primarily by the addition of new features rather than by revision. The design of the version of CMS-2 being evaluated here, CMS-2Y(7), was influenced by its predecessors, retaining most of their features and syntax, and by its target machine, the AN/UYK-7. The growth of CMS-2Y(7) has been primarily by the addition of new features. The most recent growth was caused by the current flurry over structured programming.

CMS-2 now contains most of the structured programming features common to modern HOLS: The case statement, WHILE and UNTIL conditions on iteration, block structuring, etc., in addition to those features that were considered necessary for embedded military systems: Machine code insertions, separate element compilation, compools, procedure linkage without a required monitor, mapping of structures to the machine, high level access to machine functions and hardware, and features chosen for their high efficiency (both in execution speed and space), such as the procedure switch. Some of the features make CMS-2 highly machine dependent and are contradictory to the philosophy of DOD-1.

As one would expect from a language that is an amalgam of features advocated by different schools of language design, there are some inconsistencies and context dependencies, although through careful design these have been kept to a small number. Similarly, an attempt has been made to avoid unnecessary complexity of the syntax and semantics. Although large, the language is not unwieldy or difficult to learn. In subsequent versions of the language some of the seldom used features have been culled and some of the syntax has been redefined to resolve inconsistencies and create a more uniform notation.

CMS-2 meets, at least partially, many of the Tinman requirements, primarily those that would be met by most high order languages in current use for military systems. It has a wide variety of data types and structures, including records, arrays, and fixed point, floating point, character string, and enumeration types. It has literals for most data types (not for Boolean, however!) and allows data initialization. It has all the standard arithmetic, relational, and logical operators. It has the standard control structures: Sequential, conditional, iteration, and some exception handling. It is free form, allows mnemonic identifiers and labels, has adequate commenting capabilities, and has access to libraries. It has functions with multiple inputs and procedures with multiple input and output (result) parameters.

What is missing from the language are many of the more modern features which allow a higher degree of programming sophistication. These include extension capabilities, recursion, parallel processing,

pointers, generic procedures, and suitable I/O capabilities and scope rules. The I/O capabilities in CMS-2 are very rudimentary, designed primarily for batch operation. All I/O for embedded systems is performed using machine language inserts which interface with special purpose executives tailored to the specific application. CMS-2 does not permit the scope of data units to be local to a procedure, a block, or a control structure.

It is not clear that using CMS-2 as a starting point for creation of a DOD-1 language would be the most expedient approach. It seems likely, pending more precise definitions of the features to be added, that CMS-2 could be modified to more closely meet the Tinman requirements, but the modifications would doubtlessly be extensive and costly. The resultant language probably would not be as simple and concise as desirable. Certainly it would no longer be CMS-2.

A COMPARISON OF

CORAL 66

to

TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language CORAL 66 to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CORAL 66 is considered to be defined by:

Inter-Establishment Committee on Computer
Applications
Official Definition of CORAL 66
Ministry of Defence
Her Majesty's Stationery Office
London, 1970

Tinman contains 78 language requirements. This report compares CORAL 66 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CORAL 66 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring it into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....T

CORAL 66 requires an explicit specification of the type of each variable and component of data tables. Although it allows implicit conversions during expression evaluations, the type or mode of evaluation is determined at compile time and does not change at run time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogenous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....P+

Integer	T
Floating Point	P+
Fixed Point	T
Boolean	F
Character String	P
Arrays	T
Records	T

CORAL 66 provides for integer, floating, fixed, and array type of data but does not provide for Boolean variables. The official definition of the language does not allow character data declarations, but Blandford extensions to CORAL 66 permit BYTE declarations which are used for character definitions and string manipulations. The TABLE declaration allows the definition of the type of a table's components including their width and length. This is equivalent to defining a record. The implementation of floating point is optional and is dependent upon the hardware. If the computer does not have the floating point unit, this feature is not required.

Introduction of Boolean type variables will require changes to existing implementations in type checking, table declarations, comparison operations, function and procedure parameter definitions etc. The floating point type data definitions can be made mandatory since this feature can be provided by software at the cost of some storage space. This may be significant on mini computers, but will cause

minimal problems on larger computers. The Blandford extension can be merged with the language kernel and made part of the official definition, instead of being left as an extension.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.P

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedT

CORAL 66 does not allow global specification of the precision for floating point arithmetic. In fact it permits no precision specification at all for floating point numbers. They are confined to one full word.

The language does permit individual precision specification for fixed point numbers and thus meets the second part of this requirement.

To meet this requirement the language will have to be modified in following ways: (1) Implementation of floating point data will have to be made mandatory, (2) the syntax of the language and the data definition will have to be modified to permit specification of precision by the user, and (3) global specification of precision for floating computations will have to be allowed.

These changes to the language will impact all existing implementations since none of them permits global specification of precision for floating numbers. Besides, many existing implementations do not have the floating point option at all.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.P+

Treated as exact quantitiesT
Range and step size determined at compile timeP
Scaling handled automaticallyP

The requirement implies that the user should be able to specify explicitly the range and step size of the fixed point numbers. CORAL 66 allows the user to specify the total bits and the fraction bits associated with a fixed point number, thus implicitly setting the possible range of values and the step size, but does not allow an explicit declaration to do so. Furthermore, it does not allow the number to exceed one full word of the computer, thus making the fixed point data definition and its range machine dependent. Scaling for each variable is specified by the user, but is automatically handled by the compiler while processing expressions. Thus the language only partially meets this overall requirement.

The language definition should be modified to allow the user to specify the range and step size of fixed point variables. Furthermore, the language should not restrict the size of a fixed point variable to one computer word. The user should control the size of the word and its range and not the computer word size or the compiler.

Such modifications to the language will impact all existing implementations significantly since they all provide for one computer word to define a fixed point and floating point number.

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)F

CORAL 66 does not permit definition of a user-specified character set. The character set is defined to be ISO (Same as ASCII). Furthermore, no facilities are provided in the language to alter the collating sequence of these characters at compile time. The conversion capabilities between EBCDIC and ASCII are dictated by implementations and not required by the language definition. Thus CORAL 66 fails to meet this requirement.

Introduction of a capability to enumerate a character set in CORAL 66 is a major language design change having an equally significant effect upon implementation. It would require adding status variables and enumeration capabilities which the language does not presently have.

Changes in the language to accommodate user defined character sets will require modification to all existing implementations. The library will have to include routines to convert ASCII to EBCDIC and other installation and implementation related conversion routines.

Other features of the language related to strings and character sets will also be affected to accommodate the programmer-defined character set.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P+

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryF
Subscripts only integers or from an enumeration typeT
Subscripts will be from a contiguous rangeT

CORAL 66 allows one or two dimensional arrays each element of which is of the same type. It requires that the lower and upper bounds be supplied by the user in the form of integers, and thus be determinable at compile time. The language definition does not allow the computation for the upper bound to be deferred until scope entry time. It must be available at compile time.

To defer processing of upper subscript bound implies provision for dynamic array sizes. This would mean that storage allocation for the array would be done at the run-time. It would require more time and care to execute the programs having dynamic arrays. Modifications to the translators will have to be made to alter the existing syntax-checking mechanism (which checks for the numerical value of both upper and lower bounds), the storage allocations functions (which allocate core for arrays at compile time), and the compile time diagnostics.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.P

Alternative structures for records are possibleT
Discrimination condition may be any Boolean expressionF

The OVERLAY feature of CORAL 66 causes apparently different data references to refer simultaneously to the same objects of data (i.e., as alternative names for the same storage locations.) This feature can be used to overlay, and later access, variables and array or record components. However, there is no explicit discrimination condition available to the user for accessing the record.

To meet this requirement the language will have to provide a mechanism to allow the user to evaluate a Boolean expression and then use one or the other form of record in his program.

Introduction of this feature will require changes to all existing implementations to allow the OVERLAY feature to work in conjunction with the discrimination condition.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.T

Automatically defined for any type (except...)T
Available for individual componentsT
(Assignment and reference via functions)F

Assignment to integer, floating and fixed data types is available in CORAL 66, as is the user ability to reference the values stored. Components of arrays and records can also be individually assigned to or referenced. However, the assignment operation is not defined for entire records or arrays, even if they are conformable.

No conflicts with other requirements.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.T

CORAL 66 contains = operator which can be used to compare two logically equivalent values of different types for equivalence. Thus in a logical comparison floating 1.0 will be equal to integer 1.

No conflicts with other requirements.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P-

Built-in for all numeric and enumeration typesP
Ordering can be inhibited when desiredF

CORAL 66 provides for relational operators <, <=, :=, >=, <> and > for logical comparison of numeric data. However, since CORAL 66 does not permit status variables, automatic ordering of enumerated data is not permitted. Unordered definitions of sets is also not allowed.

To meet this requirement, CORAL 66 will have to define STATUS variables with the capability to define their associated set of values either in an ordered or unordered way. Introduction of this type of variable will affect the existing implementations significantly. All logical expression processing will have to be modified. Its impact on control statements will require changes in IF and FOR-WHILE statements also.

All existing implementations will have to be modified to include the Status variables and changes to the constructs mentioned in the preceding paragraph.

B4. The built-in arithmetic operations will include:
addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.P+

Addition	T
Subtraction	T
Multiplication	T
Division with real result	T
Exponentiation	F
Integer and fixed point division with remainder	F
Negation	T

CORAL 66 does not provide for exponentiation nor does it provide for integer division with remainder. Integer division of the type $K := I/J$; results in the integer quotient being stored in K and the remainder is lost to the user. However if K were defined to be floating, then I and J will be first converted to floating before division takes place. The function INTEGER has to be used in such cases to force division before conversion (i.e., $K := \text{INTEGER}(I/J)$). Operators for addition, subtraction, and negation, with appropriate conversions where necessary, are also provided.

Implementation of exponentiation can be done via software subroutines. Many CORAL 66 compilers already have this feature. Integer division with remainder will require some syntax changes in the language to enable the user to specify where the remainder has to be stored. Both of these features can be implemented with minimal impact on existing translators. Areas impacted will be syntax analysis, expressions, diagnostics and library.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

....F

Never from the left for data within rangeF
Never on the right for integer and fixed pointF
Implicit floating point rounding beyond precision allowedF
(Run time checks can be avoided)N/A

The CORAL 66 language does not provide explicit rules or functions for truncation from the right for integers or fixed point numbers nor does it prohibit truncation from the left. This matter is left entirely in the hands of the implementors. On the floating point numbers, the user has no option to specify the precision. Each floating number occupies one computer word. Their level of precision, and the precision of intermediate results is determined by the hardware or the implementors. Hence the language fails to meet this requirement.

The CORAL 66 definition can be modified to include rules for truncation and rounding. Users should be allowed to specify precision for floating point numbers. Implementation of these changes can be accomplished with minimal effort. It is conceivable that some of the existing compilers may not be allowing truncation from the left. Other features impacted as a result of these changes will be multiplication, addition, assignment, and expression processing.

The existing implementations will have to ensure that these rules for truncation and rounding are also followed to compute and store the intermediate results.

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.

....P

Short-circuit andT
Short-circuit orT
NotF
XorF

CORAL 66 requires short circuit evaluation of the AND and OR Boolean operators. It does not provide for NOT or XOR.

The introduction of NOT and XOR Boolean operators will have minimal impact on expression evaluation and some control statements requiring logical conditions (e.g., IF, FOR WHILE etc.). The existing implementations will have to be modified accordingly.

R7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.F

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF

The CORAL 66 language definition does not specify that scalar operations be extended to conformable arrays or that data transfers be permitted between records or arrays of identical logical structure. Some implementations of CORAL 66 provide some of these facilities, but that is an accident of implementation, not a language specification.

This feature is not difficult to implement. It would require further type checking for arrays and components of records, their conformability for size and subscript ranges before generating code to perform the scalar operations or data transfers for the entire array or records.

R8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.P-

No implicit conversionsF
Explicit between integer, fixed point, and floating pointT
Explicit between fixed point scale factorsF
(Explicit between integer and Boolean)F
(Explicit between integer and enumerated types)F
(Explicit between different enumerated types)F

During expression processing the CORAL 66 language definition permits implicit type conversions. However, explicit conversion operators between integer, fixed, or floating point data are also available. Any expression may be enclosed in parentheses and be explicitly typed by a prefix 'INTEGER', 'FIXED' scale, or 'FLOATING'. For example

'FLOATING'(N + 'INTEGER'(X*Y - A/B))

is a primary of type floatina.

To introduce and further emphasize the explicit conversions in the language, the current language definition and implementations will have to be modified to forbid implicit conversion. Some modifications to the syntax-checking procedures will also have to be made to accommodate explicit conversions between two fixed-point scale factors.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

....P-

Implicit conversion between rangesF
Exception condition on integer and fixed point truncationP

There are two kinds of ranges: the range of variables and the range of subscripts. CORAL 66 does not permit specification of range of variables, hence the question of implicit conversions between ranges does not arise in this case. Array size ranges are required in CORAL 66, but they must be supplied by the user as integer constants and are compile time evaluable. The definition seems to imply that an error message will be issued to the user if he supplies other than an integer constant as a subscript range. No automatic conversion is implied.

The language does not require that an exception condition be raised if an integer or fixed point number is truncated.

The language does, however, permit a run-time mode of operation in which exception conditions are raised if the subscript range is exceeded.

To meet this requirement CORAL 66 compilers may have to be modified to allow subscript ranges of types other than integers, and also to permit automatic conversions or truncation of these ranges to integers. The concept of ranges for all variables will have to be introduced and the language syntax modified to accomplish this.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

....F

Sending and receiving of dataF
Sending and receiving of control informationF
Dynamic device assignmentF
User exception condition controlF
Installation independenceF
(Data formatting capability)F
(Reading and writing of bit strings)F

The CORAL 66 language makes no provision for I/O statements as part of the official language definition.

Language features to select a device, open and close a file, read and write data, handle interrupts, etc. should be defined. An I/O package, similar to the one currently available on the GE4080 computer at the Royal Radar Establishment, should be used (with necessary modifications) to carry out the functions required by the I/O statements.

B11. The language will provide operations on data types defined as power sets of enumeration types (see F6). These operations will include union, intersection, difference, complement, and an element predicate.

....P-

UnionT
IntersectionT
DifferenceT
ComplementF
Membership predicateF
(Set inclusion)F

CORAL 66 permits the operations of DIFFER, UNION and MASK to perform the logical operations of difference, union and intersection respectively. It does not provide for other logical operations specified in Tinman requirements. These operations are only defined for the existing types since CORAL 66 does not allow user-defined data types.

CORAL 66
Requirement B11

12

To fully meet this requirement the language will have to (1) allow user definition of new data types and (2) allow all of the powerset operations specified above to be applicable to both the built-in data types as well as the user defined data types. This would require major changes to the current definition of the language as well as to the existing implementations.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.P-

Side effects must occur in left-to-right orderP-
(Embedded assignments)F

CORAL 66 definition does not specify algorithms for expression evaluation and leaves their implementation dependent. Only for evaluation of library built-in functions in a simple expression does the language definition state that they will be evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets. The only exception to this rule is when other functions appear as value parameters of a function and hence must be evaluated before the function is evaluated (i.e., the order of evaluation of $\sin(\cos(\exp(n)))$ will be exp, cos, sin.).

To establish uniformity in all implementations, the language definition should specify rules for expression evaluation. Only where the order of evaluation is important should the language impose the restriction that evaluation must proceed from left to right. In other instances the compiler implementors should be given the choice in selecting evaluation rules, as long as the effect and the results of evaluation are not changed.

Such a change is likely to affect the existing implementations which have followed their own rules for the order of evaluating expressions.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.P

Few precedence levelsF
No user-defined precedence levelsT
Operands of an operation are obviousT

CORAL 66 does not define precedence levels for the operators. The only established precedence rule is that all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. If an expression is enclosed in parentheses, its terms are not outermost and the rule no longer applies. The algorithm for the particular compiler will determine the sequence of events.

A few precedence rules for operator hierarchy can be established in the language definition. This may affect some, though not all, implementations. For instance, one set of precedence rules can be: 'MASK', 'UNION', 'DIFFER', '/', '+', and the ordinary priority of operators may be changed by using parentheses within an expression.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

....T

The language allows expressions wherever both constants and variables are allowed.

No conflicts with other requirements.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

....F

The official definition of CORAL 66 does not permit constant expressions wherever constants are allowed. For instance, FIXED(2*8,3/1) X,Y; is an illegal data definition, whereas FIXED(16,3) X,Y; is acceptable. Similarly, the lower and upper bounds of the array in an array declaration must be a constant, and not a constant expression.

If a requirement is established in the language definition that all constant expressions must be evaluated before run time, then CORAL 66 will meet this requirement. Constant expressions, such as in the examples above, can then be legally used. This will require minimal changes to syntax checking routines of existing compilers.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to

ease of learning.

....P

Parameter rules consistent in all contextsF
No special operations applicable only to parametersT

CORAL 66 allows procedure parameters, parameters for declarations, and built-in operators, but does not permit parameters for exception handling or parallel processing. While procedure parameters can be objects of data, places in the program, data references, procedure names, expressions, etc., the parameters in several data declarations (e.g., FIXED option) can only be integer constants. Hence the language is not uniform in its treatment of parameters in the sense of the Tinman requirements. However, no special operations are allowed on parameters which are also not permitted for other structures.

The language will have to provide for exception handling control (or other) features, for parallel processing and for user definition of new operators. These are all major changes to the existing language and will require significant effort to implement.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.P

Actual and formal parameters will agree in typeT
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedF

CORAL 66 requires type checking for compatibility between actual and formal parameters. It also requires that the number of dimensions, the size, and the subscript range for array parameters be known at compile time. Thus it does not allow for dynamic arrays and hence does not meet part of this requirement.

The language definition must be modified to allow for dynamic arrays and also passing of parameters related to the size and subscript ranges of these arrays. This would be a significant change for existing implementations.

C7. There will be only four classes of formal parameters.

For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P+

Act as constants (call by value plus)T
Act as variables (call by reference)T
Exception controlF
Procedure parametersT
(Act as variables, but call by value)T
(Act as variables, result parameter)T

CORAL 66 provides for all of these types of parameters with the exception of exception handling parameters. There is no provision for exception handling in the language. However, it also provides for parameters not required by the Tinman requirements (e.g., SWITCH).

The exception handling mechanisms should be added to language features (e.g., ON OVERFLOW etc.) which allow transfer of control to labels outside the scope in which the exception condition occurs. Those types of parameters (e.g., SWITCH) which are not required by the Tinman requirements should be disallowed.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.P-

Above properties optionalF
Above properties are fixed at run timeP

CORAL 66 does not support the capability to define generic procedures in which the type of the parameters is determined at compile time by the type of the actual parameters. The type specification for each parameter on the formal side is required in CORAL 66 and is not optional as specified by the Tinman requirements. The same is true for dimensions of the arrays and scale factors of fixed point variables. CORAL 66 does not permit specification of range, precision, or format in parameters.

Implementation of this requirement will imply many changes in the existing calling mechanism of the language. Each parameter, for instance, will be followed by type specification, a range of values, precision specification where applicable, dimensions, and scale in the

calling mechanism. Some of these items are not even currently defined in the language. Additionally checking for these items will have to be implemented on both the actual as well as the formal side of the call. On the formal side the generic procedure capabilities will have to be introduced which will require selection and/or generation of code for the formal procedure for the type, mode, precision, etc. required by the actual calls.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.F

Variable number of arguments possibleF
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeF

This facility is not available in CORAL 66.

Implementation of this capability will affect the language and all implementations. It will require provision for variable number of parameters in procedure calls. Even the existing I/O procedures in some of the existing implementations will have to be modified to accommodate this.

D1. The user will have the ability to associate constant values of any type with identifiers.F

CORAL 66 does not permit an identifier to be associated with a constant, except by assignment.

Minimal changes to the language definition and existing implementations will be required to allow a variable to be declared a constant and be associated with the value of a constant of the same type.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesT
Consistent interpretation in program and dataF

The CORAL 66 definition provides for the syntax for the built-in data types e.g., fixed, floating and integer. Blandford extensions to the language also include definition and syntax for characters. But the language does not provide for I/O features nor does it specify the syntax for data to be input or output.

Several implementations (maybe all existing implementations) of CORAL 66 allow the same value for numeric constants within the program and I/O data. To meet this requirement, standardized I/O features have to be included in the language definition. Minimal changes to process those I/O statements utilizing existing I/O routines and facilities will be required in the existing implementations.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P+

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesF

CORAL 66 allows initialization of variables at the time of their declaration. Initialization of these variables takes place at the time of scope entry. However, there is no language specified guarantee that no default initial values will be assigned to variables not explicitly initialized by the user. Nor does the language definition specify that an error condition will be raised if a uninitialized variable is accessed before assignment of a value. The subject is implementation dependent. Hence CORAL 66 does not meet this portion of the requirement.

All existing implementations will be affected if they are to meet this requirement. It will require provisions for compile time checks to ensure that no variable is being accessed without first being given a value either by assignment, I/O, or initialization. Error diagnostics should be given if this is not the case. Furthermore, no implementation should initialize variables by default, as is commonly done.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....F

Numeric variable range specification mandatoryF
Fixed point variable step size specification mandatoryF
Range and step size specifications do not define a new typeN/A

CORAL 66 does not allow range specifications for numeric variables nor does it permit step size specification for fixed point variables.

Implementation of this feature will require modification of CORAL 66 syntax to allow explicit range specification for all variables and step size specification for fixed point variables. The compilers will then have to generate code to check for ranges at execution time.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

....P-

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataP-

CORAL 66 does not meet the first part of this requirement because it does not permit specification of any kind of ranges.

In the second part of the requirement, there is no way that records can be made part of an array nor is it conveniently possible to make arrays part of a record. However, in table declarations (which represent records) one can define an element of an array as constituting one entry in the record. Subsequent entries in the table can be made to represent subsequent elements of the array. Only in this partial sense can an array be made a component of a record.

To completely meet this requirement would require changes in the language and the implementations. The language has to be modified to accept ranges in declarations, definition of new data types, status variables, and an expanded definition of records and arrays, to permit the inclusion of arrays in record definition and vice versa.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....P-

Recursive and network structures providedF
Handles variable-value and structure-component connectionsP
Pointer property is an attribute of a typed variableF
Pointer property not for constants, affects only assignmentT
Pointer property mandatory for dynamic allocationF
Allocation scope never wider than access scopeT
(Either the value or the pointer is modifiable)T
(Pointer mechanism handles procedures and parameters)P
(Remap and replace assignment have different syntaxes)F
(Built-in dynamic variable creation)F
(Variable equivalence classes are declarable)F

CORAL 66 does not provide for a pointer data type. However, its LOCATION operator can be used to point to the location containing data values for a variable or array and to access those values. The language does not provide any safety features for this option and the programmer himself has to ensure the validity of the pointers and the data being accessed. If the pointer is pointing to the first item in a table or array, items in the subsequent words can be referenced; but if the item

is located in a part of a word, there is no guarantee that it can be referenced.

Although it is possible to structure lists using the LOCATION option, the CORAL 66 language does not permit declaration of such lists in the data definition. The LOCATION option is also inadequate to define recursive data structures.

To properly meet this requirement, the language must provide for a pointer data type capable of pointing at variables, arrays, tables, elements of tables, procedures, parameters etc., and to allow recursive data definitions and null references. It must allow for easy modification of the data value pointed to or the pointer value itself. Such changes in the language will affect all existing implementations.

E1. The user of the language will be able to define new data types and operations within programs.F

CORAL 66 does not provide for user definition of new data types or operations.

Special type checking mechanisms will have to be added to existing compilers to allow for recognition of new data types and operations and then their use as built-in types. The language definition will have to be modified to show how new data types and operators can be added and used.

E2. The "use" of defined types will be indistinguishable from built-in types.F

Not applicable to CORAL 66 since it does not allow defined types.

The changes needed to fulfill this requirement are discussed under requirement F1.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.T

CORAL 66 requires that every data item be explicitly declared before use.

No conflict with other requirements.

E4. The user will be able, within the source language, to extend existing operators to new data types.F

CORAL 66 does not allow defined data types and hence does not provide the capability of extending the existing operators to defined types.

AD

-A037 640 COMPUTER SCIENCES CORP FALLS CHURCH VA
DOD PROGRAM FOR SOFTWARE COMMONALITY HIGH ORDER LANGUAGE WORKIN--ETC(U)
1977

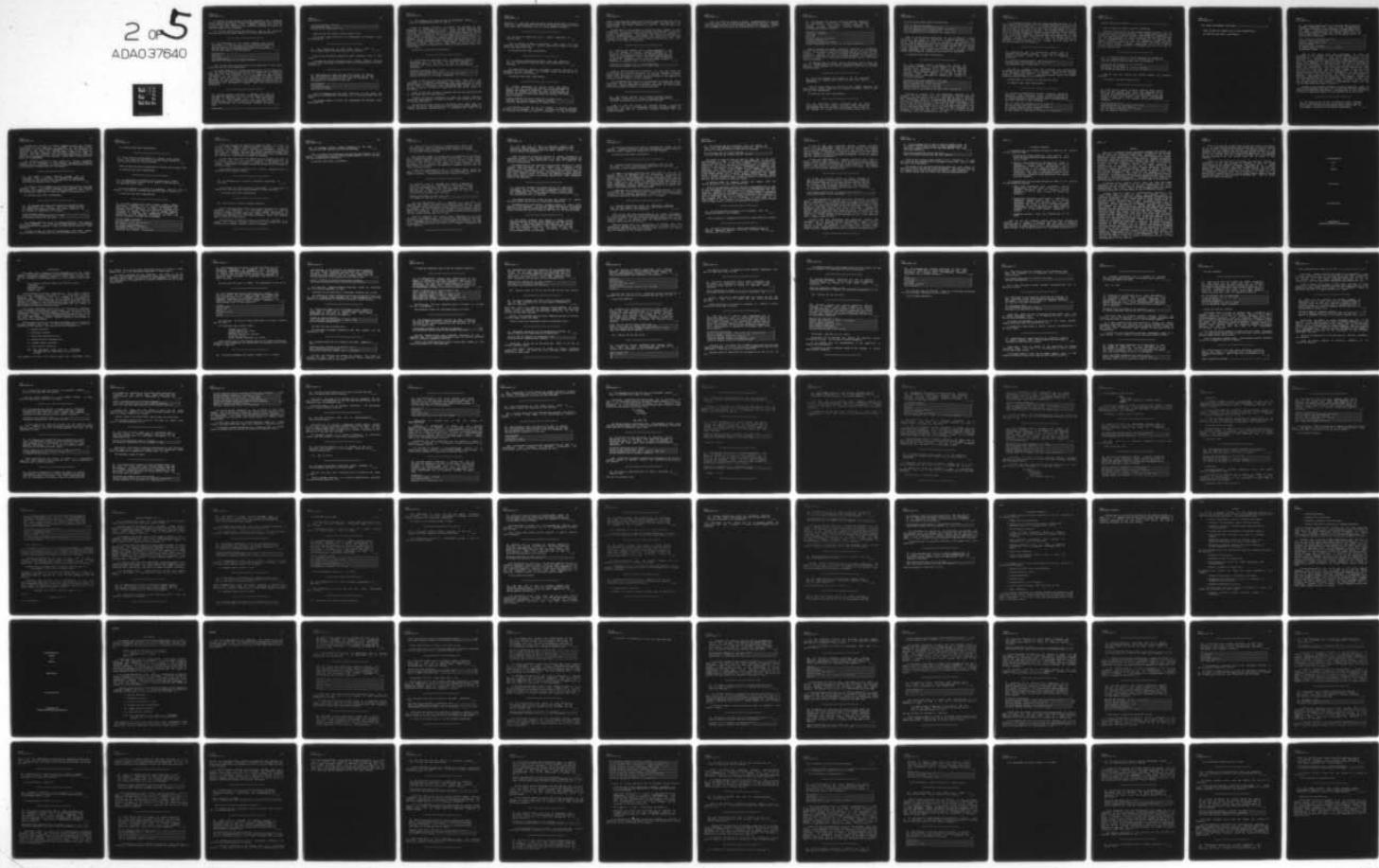
F/G 9/2

N00039-75-C-0289

NL

UNCLASSIFIED

2 of 5
ADAO 37640



In addition to definitional facilities specified in E1, a mechanism should also be provided to specify the operations that can be performed on the defined data types. Other language features will also be affected since their definition will have to be extended to provide for new data types and new operators.

All of these modifications specified in E1 and in the preceding paragraph will significantly impact all existing implementations.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.F

ConstructionF
SelectionF
PredicatesF
Type conversionsF
Operations and data can be defined togetherF

CORAL 66 does not allow definition of new operations or data types, hence does not meet this requirement.

The entire capability to extend the capabilities of the language to meet the needs of different applications by allowing user to define new data types and new operations has to be added to the language. Furthermore, the user should be able to associate new operations with new data types and should also be permitted to extend existing operations to new data types. This would require changes in existing translators in the area of lexical and syntax analyzer, construction of dictionaries and intermediate language table, and the type checking mechanisms.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.F

EnumerationF

Cartesian products (records)F
Discriminated unionF
Powerset of an enumeration typeF

CORAL 66 does not support defined types at all.

The changes needed to fulfill this requirement are discussed under requirement E1.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.F

CORAL 66 has an overlay capability, which provides a form of free union.

To remove the overlay capability would greatly simplify existing compilers, but would have a profound effect on existing programs written in CORAL 66.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.F

InitializationF
FinalizationF
Allocation actionsF
Deallocation actionsF

Since the language does not allow definition of new types, the questions of initialization, allocation, etc. of new data types does not arise.

The changes needed to fulfill this requirement are discussed under requirement E5.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.T

CORAL 66 allows allocation of storage for variables at the beginning of procedures, in a block (e.g., between BEGIN and END statements) and via a COMMON statement. Access to the variables is controlled by language defined rules. A variable allocated by a COMMON statement can be accessed by several procedures having that variable in their COMMON declarations; a variable declared at the beginning of a procedure can be accessed from anywhere within the procedure except in embedded macros or blocks containing another declaration with the same name. A variable declared within a BEGIN and END block will only be accessible within that block. Thus CORAL 66 fully meets this requirement because it separates the scopes of allocation and access for defined data.

No conflicts with other requirements.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.P-

Allowable operations can be limitedF

Access can be limited where usedF

External declarations need not all have the same scopeT

Naming conflicts can be avoided (renaming)F

CORAL 66 does not allow specification of access rules during the definition of structures (procedures, data etc.), nor does it allow specification of access rules at the place where the structure is used. At the time of structure usage language-defined access rules are followed in case of a conflict of names, but the user has no choice in the matter.

CORAL 66 also does not have a renaming facility for cases where two or more names refer to same location.

However, the current definition of CORAL 66 permits different scopes for items declared in COMMON statement. This is done at the time of variable definition.

The ability to limit access to structures both where they are defined and where they are accessed will require extensive changes in language syntax and implementations. A mechanism will have to be

specified to state the scope and access rules at the time of structure definition. Implementations will have to be modified to test for access rights each time a structure tries to access another structure.

F3. The scope of identifiers will be wholly determined at compile time.

....T

This is required in CORAL 66 definition. Scope rules have been specified and resolution of apparent conflicts in names is resolved before code generation time.

No conflicts with other requirements.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

....T

The capabilities to develop application oriented libraries are available in the language. The LIBRARY option permits the use of procedures stored in the library.

No conflict with other requirements.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

....P

Program component libraries accessible at compile timeP
Libraries can contain foreign language routinesT
Interface requirements checkable at compile timeF

The libraries of CORAL 66 are not capable of holding anything definable in the language. For example, it cannot hold data definitions by themselves unless they are parts of a procedure. The use of EXTERNAL

option allows interfaces with object programs compiled from other source codes provided they are compatible with the loader. Thus the check for compatibility with these routines is made at load time and not at compile time.

Modifications to the language to define and store groups of data definitions in the library for later access by procedures is necessary to meet this requirement. This facility will have to be included in existing implementations. Furthermore, the existing implementations will have to be modified to allow compile time checks with object code compiled from other source codes instead of at the load time, as is specified in the CORAL 66 definition.

F6. Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.P-

Libraries and compools will be indistinguishableF
Immediately accessible sublibraries at any levelP

CORAL 66 does not provide for compools. It does not allow for partitioning of libraries according to use by different projects although the effect can be achieved by the existing library facilities which allow any number of project oriented routines to be compiled and stored for later access by project programs.

The implementations will have to be modified to include the compool facilities and to make them compatible with the existing library facilities. Simple procedures must also be developed for partitioning libraries according to general or specialized use.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.F

CORAL 66 does not provide for standard machine independent capabilities for I/O, file management, or other peripheral usage. No attempt has been made to standardize language interface with the operating system.

This is a difficult feature to define. Standardization of language interface with the OS is still beyond the state-of-the-art. However, some language features can be defined to perform I/O functions. Changes will be required in the existing implementations to accommodate this.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.P

Sequential execution	T
Conditional execution	T
Iteration	T
Recursion	P
(Pseudo) parallel processing	F
Exception handling	F
Asynchronous interrupt handling	F
Control structures from a small set of simple primitives	T

CORAL 66 permits the normal sequential execution of statements, provides for IF-THEN-ELSE conditional control and FOR loop for iteration control. It optionally allows the RECURSIVE option to be specified for procedures. However, it does not provide for pseudo or real parallel processing, exception handling, or interrupt handling. The language has defined a small set of primitives which have been used to build these control structures.

The language needs to define control structures which allow the user to perform parallel processing, exception handling, and interrupt handling. These features will have a significant impact on all existing implementations.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.F

CORAL 66 allows labels to be defined in the COMMON statement and thus permits global jumps from one segment to another. Hence it does not meet this requirement.

No conflict with other requirements.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a

computed choice among labeled alternatives.Pt

Based on Boolean expressionT

Based on type from discriminated unionF

Based on computed choice among labeled alternativesT

All alternative must be accounted forT

Simple mechanisms will be supplied for common casesP

CORAL 66 allows Boolean expressions in its IF-THEN-ELSE control structure. The result of evaluation of Boolean expression in short circuit mode determines the action to be taken. The language also allows a SWITCH statement in which the control is transferred to one of the labelled variables depending upon the value of the index but not the type of the discriminated union. The language requires all alternatives in the conditional controls properly accounted for. It also permits simple mechanisms for common cases by allowing, for example, a short circuit mode evaluation of Boolean conditions in IF statement.

No conflicts with other requirements.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).Pt

Termination can occur anywhere in the loopT

Multiple terminating predicates are possibleT

Entry permitted only at the loop headT

Simple cases are clear and efficientT

Control variable is local to the loopF

Control value is efficiently available after terminationT

The CORAL 66 requires that the termination condition for FOR statement in both forms (i.e., FOR with STEP or FOR with WHILE) be provided at the beginning of the loop. Each time the loop is executed, this termination condition is evaluated at the beginning of the loop. The user is allowed to use an IF-THEN or GOTO statements anywhere in the loop giving him the facility to terminate the loop. The language also allows a loop to be terminated in more than one way, permits entry only at the top and allows simple forms. The loop control variables are not local to the loop, they must be declared in the beginning of the procedure in which the loop occurs. The value of the control variable is available after the termination of the loop.

The requirement stating that the control variable be local to the loop seems to be in conflict with the requirement that all variables be explicitly declared before use. If a control variable is defined before use in a loop, it will also be defined after the termination of the loop and thus will not be local to the loop. For instance, CORAL 66 requires that all control variables be defined at the beginning of the procedure in which the loop is present. These variables are not local to the loop. On the other hand, the assumption that the construct FOR I=1 THRU 5 defines I is also wrong. FOR I=1 etc. does not really define the construct variable I. It assigns a value to the variable without defining it. The attributes of I are unknown and will have to be defined by default by the compiler.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

....P

No recursive procedures within recursive proceduresF
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)T

The CORAL 66 language allows both recursive and non-recursive procedures in the definition. If a procedure is not explicitly declared to be RECURSIVE, non-recursion is implied. There is no language specified restriction that a recursive procedure cannot have another recursive procedure within its definition.

Maximum depth of recursion is also not specified.

No conflict with other requirements.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

....F

Able to create and terminate parallel processesF
Process can gain exclusive use of resourcesF
No parallel routines within recursive routinesF
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF

(Access rules are enforced)F

CORAL 66 does not support parallel processing.

Extensive modifications to the language capabilities will have to be provided to meet this requirement. The language syntax will have to include facilities for task initiation, termination and resumption and will also have to provide for waiting, event related interrupt handling, etc. The implementation of these facilities in the compiler will require extensive interfaces with the scheduler, loader, interrupt handler, storage allocator, disc and file handler, and other portions of the operating system. Hardware capabilities (e.g., the available storage, single or multi-processing environment etc.) will also have to be taken into account.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.F

Program can get control for any exceptionF
Parameters can be passedF
Can get out of any level of a nest of controlF
Can handle the exception at any level of controlF

CORAL 66 does not include any special support for exception handling.

No conflict with other requirements.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.F

Priority specificationF
Synchronization via wait/enable operationsF
Wait for end of real time intervalF
Wait for end of simulated time intervalF
Wait for hardware interruptF

(Can enable and disable interrupts)F

CORAL 66 does not support any of these capabilities.

No conflict with other requirements.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....P+

Free format with statement terminatorP+
Mnemonic identifiers possibleT
Based on conventional formsT
Simple grammarT
No special case notationsT
No abbreviations of identifiers or keywordsT
Unambiguous grammarP

CORAL 66 very nearly meets this requirement. It allows its statements to be entered in free format anywhere in the line and terminated by a semi-colon. There are a few exceptions to free formatting e.g., no blanks are permitted in composite operators like >= etc. The language does not limit the length of the identifier names, but allows the compilers to optionally select the first 12 characters to determine the uniqueness of the identifiers. The user can thus keep the names more mnemonically significant, and the compilers can keep their syntax checking efficient. CORAL 66 syntax follows conventional forms and left to right scan and has simple grammar rules such as the parenthesis are evaluated first in any expression. It does not have special notation for rare cases, nor does it allow abbreviations of keywords in the language definition. The syntax of the language is generally unambiguous with few exceptions. For example, example, the two constructs BIT and BITS are syntactically very close to each other and yet have significantly different usage. Similarly, the constructs AND and MASK have the same meaning in mathematical sense, but have significantly different usage in CORAL 66.

Changes to the language definition to select constructs which are significantly distant from each other to represent different concepts is necessary to meet this requirement. Implementation of such changes will be trivial although the impact on user programs may be significant.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

....P

Although the facility to modify language syntax and operator hierarchies or the capability to introduce new precedence rules, definition of new infix operator precedence, etc. is not available or permitted to the user, the language does allow use of additional features (including keywords) not officially within the CORAL 66 language, and not clashing with the official definition or with each other, to be implemented in the compiler. Hence CORAL 66 only partially meets this requirement.

The language definition should explicitly forbid additional keywords to be introduced in the language. Extensibility should be permitted only via procedure, user data and operator definitions without modification of existing rules of the language.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

....T

All features of the language can be constructed utilizing the ASCII character set. In certain instances the language has provided concessions to accommodate representations acceptable to ASCII (e.g., the conventional "less than or equal to" symbol is replaced by <=).

No conflicts with other requirements.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

....T

Break character exists
(Literals are self-identifying as to type)T
(Bit-string literals for any type)P

The language specifies rules for forming identifiers and literals and gives examples. It allows use of blank character in identifiers without acting as a terminator. Hence the blank can be used as a break character.

The type of CORAL 66 literals is determinable from their syntax. Octal literals are allowed, but bits are not permitted as literals.

No conflict with other requirements.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.T

CORAL 66 does not allow continuation of lexical units across lines.

No conflict with other requirements.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.T

A list of keywords is provided in an Appendix. There are only 44 keywords. They are all reserved and cannot be used as identifiers.

No conflict with other requirements.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.P+

Uniform comment conventionT
Look different from codeT
Bracketed by one or two charactersP
Can contain any charactersP
Can appear anywhere reasonableT
Terminated by the end of the lineF
Compatible with automatic reformattingT

There are two ways of inserting a comment in CORAL 66: First, by starting with COMMENT and terminating with a semi-colon. In this case the text of the comment cannot contain a semi-colon. Second, by enclosing the comment within round brackets immediately following a semi-colon in a program. The text may contain brackets provided they are matched. In either form of comment, the end of line is ignored in the comment and a semi-colon terminates the comment.

To meet this requirement the COMMENT form of comment shall have to be dropped since it does not start with one or two characters. The language should allow any character, including a semi-colon, in the comment. Furthermore, the comment should be allowed to be terminated by the end of line.

Some changes will be required to all existing implementations to accommodate the language changes.

H8. The language will not permit unmatched parentheses of any kind.T

CORAL 66 does not allow unmatched parentheses in expressions or comments nor does it permit unequal numbers of BEGINS and ENDS.

No conflict with other requirements.

H9. There will be a uniform referent notation.F

CORAL 66 encloses array subscripts in square brackets and function parameters in parentheses. It, therefore, maintains a syntactic difference between function call and array element references. In addition, it does not allow functions to appear on the left hand side of an assignment statement as is permitted to arrays. Hence the language does not meet this requirement.

Implementation of identical syntax for arrays and functions is a simple fix. However, allowing functions to be treated like pseudo variables would require moderate effort to implement.

H10. No language defined symbols appearing in the same context will have essentially different meanings.T

CORAL 66 symbols are unambiguous and have the same meaning in all usages. In particular CORAL 66 differentiates between := and = and does not allow parenthesized parameters.

No conflict with other requirements.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.P

Most defaults related to program logic are specified in the manual and are under programmer control with few exceptions. For example, the SWITCH statement does not specify the OUT or default clause which determines the transfer of control in the event the value of the switch is either zero or exceeds the number of labels provided in the declaration. This feature is implementation dependent and can alter the flow of program control.

To meet this requirement the OUT or OTHERWISE clause should be added to the SWITCH statement, allowing the programmer to specify the default condition. Implementation of this feature is trivial.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.P+

Defaults specified for don't care casesT
Programmer can override the defaultsP

There are some defaults that a programmer may care about and override, but there are some others which he cannot control. In the first category comes the subscript range checking which is not checked by default. Put the programmer has the option to specify that code be generated to check it. In the second category is the parameter checking default for procedures. No run time parameter checking is performed and the user cannot specify that this be done.

CORAL 66 compilers have placed prime importance on run time efficiency and compact code. The language has, therefore, deliberately avoided features such as dynamic arrays and run time checking. To meet this requirement, an option should be provided to the user so that code is generated to perform run time checking of procedure parameters and other features.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.F

CORAL 66 does not provide the facility to provide information to compilers regarding the characteristics of the object machine, the configuration, etc. so that appropriate code could be generated for various configurations.

Implementation of this capability would require selection of these "compile time variables" for the language and processing them to generate different types of code for various computers, and configuration. A determination will also have to be made as to where to place these features in the program. Implementation of these features will have minimal effect on syntax and semantic analysis phases of compilation, but will have a significant impact on the code generation phase of the compilers.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.F

The present definition of CORAL 66 does not provide for control structures which can be used for conditional compilation.

Implementation of such control features would not impact other features of the language. The existing implementations will be affected since they will have to be modified to process the object environment related conditionals and then generate code only for the selected path.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.T

The official definition of CORAL 66 represents the kernel of the language. Blandford extension and other application oriented libraries have been added to expand the capabilities of the language.

No conflicts with other requirements.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

....P

The CORAL 66 language definition sets some limits on some of its features which are dependent only on translators. For example, it requires only one or two dimensional arrays, sets the length of the identifiers to be twelve characters (the extra characters are legal but will be ignored by the compiler) etc. However, it does not specify many other limits e.g., the number of permissible nested loops, number of nested parentheses levels in expressions, maximum number of formal and actual parameters, maximum number of identifiers in a program, etc. Hence it only partially meets the requirement.

To implement this change would require a careful analysis of user requirements and then setting a limit on these various features of the language in the official definition. Effect of such standardized limitations on existing implementations will be minimal.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....T

CORAL 66 does not impose restrictions on the object environment. It does not state limits on storage requirements, types of peripherals, real-time clock etc. In fact, it allows language features such as the floating point to be omitted from the implementation if adequate hardware facilities are not present.

CORAL 66 should make the availability of floating point data computations mandatory for all implementations, as required by A2. If the computer does not contain the floating point hardware, the option should be supplied by the software package.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.T

No efficiency cost for unused featuresT
Efficient code can be produced for all featuresT

Efficiency in terms of code and time has been one of the major considerations in CORAL 66 design and implementation. Conceptually, CORAL 66 compilation is a one pass process. The insistence that identifiers are fully declared or specified before use simplifies the compiler by ensuring that all relevant information is available when required. The syntax of the language is transformable into one-track predictive form, which enables fast syntax analyzers with no back-tracking to be employed. Features which require elaborate storage in the object machine for efficient program execution, for example dynamic storage allocation, are not included in the language. Unless run in a special diagnostic mode, a CORAL 66 compiler is not expected to generate run-time checks on subscript bounds.

In general, CORAL 66 language features are simple, clear and concise and permit efficient code generation.

There is no potential conflict with other requirements but to meet all the requirements, definition will have to be significantly modified. In order to provide for range checking on variable subscripts (A6, D4), parameter checking on procedures (C6), variable numbers of arguments (C9), user definition of new data types and operators (B11), addition of several other data types (e.g., Boolean, status etc.) the complexity of the language will have to be increased. In some cases it is questionable if new features can be added to the existing language syntax without loss of clarity and dependent efficiency.

J2. Any optimizations performed by the translator will not change the effect of the program.T

This is implied in language definition but never explicitly stated.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.F

In order to meet this requirement machine language insertions should only be permissible within the body of compile time variables (J4) in an encapsulated form. CORAL 66 does not provide for compile time variables. It permits easy insertion of machine code between BEGIN and END statements. Therefore, at present, machine code can be easily inserted within the body of executable statements. Hence the language does not meet this requirement.

To meet this requirement the language definition will have to define "compile time variables", their syntax and semantics and use. Procedures for defining characteristics of the hardware and insertion of machine code should be defined also within their body. The existing syntax for machine code insertions will have to be dropped. This change will impact all existing implementations.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.P

Encapsulated specification of representation possibleP
Space/time tradeoff can be specifiedP

The TABLE declarations in CORAL 66 allow a programmer to control object representation of composite data. The user has the facility to specify table size, word and bit position for elements and their length, and even the bit configuration while presetting the values in these tables. The language also allows the user to pack data in tables, thus permitting him some control over space utilization. Avoidance of RECURSIVE option in programs also saves core space. Not specifying the special mode of operation which allows generation of code for run time checking of subscript ranges, the user can also control some time and space also. Other features of the language, however, do not allow the user to control space or time.

To fully meet this requirement some additional "compile time variables" can be provided directing the compiler to save (1) core or (2) time. The user should be able to specify dense, medium or rare packing of data, or avoidance of excessive code to permit faster execution time. Implementation of these optimization variables will effect existing language definition and all existing implementations.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.F

Open/closed properties can be specifiedF
Open and closed versions have the same semanticsF

CORAL 66 only allows closed routines in its definition. It does not allow in-line procedures to be included in user code. Hence it does not meet this requirement.

The syntax of the procedure definition and/or the call should be modified to allow for open or closed procedures. The user will exercise his judgment as to when he wants to use open or closed procedures. It is a simple change, but will affect all existing implementations.

Extraneous Features

We recommend that the following features of CORAL 66, not required by the Tinman, be kept:

- * Parameterized MACRO capability, with nesting. This gives the functionality of open subroutines, as called for by Tinman.
- * Explicit type conversions of any data unit (typed or untyped). This can be used to control the scaling on intermediate fixed point calculations (e.g., `FIXED(13,5)(ABC - XYZ)` specifies that the result of the subtraction is to be treated as though it had 5 fractional bits) and to override any implicit type conversions. The latter consideration would no longer be valid if the implicit type conversions were removed from the language.

We recommend that the following features of CORAL 66, not required by the Tinman, be deleted:

- * Table word referencing and presetting, without consideration of the table structure. This is too hardware dependent and can seriously inhibit portability. It is also a means to defeat type checking.
- * "Anonymous" referencing, which gives a capability of addressing machine words. This is too hardware dependent, defeats type checking, and its only legitimate use seems to be as a rudimentary pointer mechanism. (It is easy to think of some appalling illegitimate uses.) Presumably this feature would be unneeded if a full-blown pointer capability is present.
- * Statement switches. These are elaborations of the `GOTO`.

CORAL 66 has one hardware feature which should probably be retained, but it should be required that each use be bracketed (encapsulated) by statements which point out the machine dependency. That feature is word logic, i.e., bit manipulation of data using the operators `MASK` (and), `UNION` (inclusive or), and `DIFFER` (exclusive or).

Summary

CORAL 66 is a general purpose programming language well suited for use on small and medium sized computers. The kernel of the language provides facilities to perform fixed point computation, one or two dimensional array manipulation, short circuit evaluation of Boolean operations, mechanisms for explicit conversions of untyped expressions to integer, fixed, or floating point numbers, type checking and matching of formal and actual parameters, call by value and call by reference, literals for all language defined data types, capability of initialization of variables at the time of their definition, a 'location' option which can be used as a pointer mechanism, and fixing the scope of identifiers at compile time. CORAL 66 allows interfaces with assembly language routines. It provides for an optional definition of recursive procedures. The syntax of the language allows free formatting, is simple and permits the user to specify mnemonic names of any length. (The translator is required to check only the first eight characters to establish uniqueness.) The smallness of the language and the simplicity of its constructs allow efficient code to be generated.

In addition to the language kernel, there are some well-known extensions to the language, such as the Blandford extension. This extension provides for bit, byte, or character definitions and string manipulation. The library option of the language permits application oriented procedures to be available for use by different projects.

However, there are a number of characteristics required by the Tinman which CORAL 66 does not have. For instance, there is no provision for parallel processing, exception handling, input/output, or real-time processing. The language does not allow the user to define new data types or new operators, nor does it allow specification of operations for built-in data types. Enumeration types are not supported. Boolean type data is not supported. No provision is made in the language for specification of global precision for arithmetic computation. The floating point data type has been made optional in the language, and the exponentiation operation is not defined. The language does not require that rounding be explicit nor does it specify that the truncation must take place from the right in all computations. The language permits implicit type conversions and mixed mode arithmetic (which can be interpreted as a case of implicit type conversion). Furthermore, the language syntax does not allow specification of the ranges of variables nor are there language defined rules for operator precedence or expression evaluation. There is no provision in the language to define and store data in compools. The SWITCH instruction provided in the language does not support structured programming practices because the control does not return to the statement following the SWITCH. The SWITCH statement also does not provide for an OTHERWISE clause. The GOTO statement is unrestricted and allows transfers of control to labels outside the procedure in which the GOTO occurs. No conditional compilation facility is available to generate code for only the selected path of conditional control statements. Nor is there a facility to allow a user to specify if he wanted an open or closed implementation for his procedures.

There is no easy way to make CORAL 66 meet the Tinman requirements. To do so would require deletion and modification of existing features and the addition of several new features. The modified version of the language will look nowhere near the existing version of CORAL 66. User resistance to change will also be significant because the new language may be so distant from the existing version that upgrading of existing programs to the new version might not be simple or even possible.

However, if the objective of upgrading the existing programs is discarded and the new language is designed to meet the Tinman requirements using selected features from CORAL 66 as a kernel (e.g. procedure definition and passing capabilities, the recursive option, FOR loop, DEFINE capability, PRESET option, explicit type conversion, explicit declarations, etc.), then it is possible to design an extensible language which has similarities to CORAL 66. The similarities would be only of a fairly general nature, however, and the new language would be quite different from CORAL 66. Such an effort would be of major proportions.

A COMPARISON OF

CS-4

to

TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language CS-4 to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, CS-4 is considered to be defined by:

CS-4 Language Reference Manual and Operating System
Interface
Intermetrics, Inc.
Cambridge, Mass.
October, 1975

Tinman contains 78 language requirements. This report compares CS-4 to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of CS-4 which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring it into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.T

The CS-4 term for "type" is "mode". the requirement is met (p. 1).

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.Pt

Integer	T
Floating Point	T
Fixed Point	P-
Boolean	T
Character String	T
Arrays	T
Records	T

Not fully met. CS-4 has no fixed point type; it has only fraction (pp. 27, 58ff.)

CS-4 does have the following types:

Integer (pp. 37-44)
Floating point (pp. 44-58)
Boolean (pp. 31-36)
Character string (pp. 100-113)
Array (pp. 82-93)
Record (called STRUCTURE; pp. 94-99)

Adding fixed point to the language would be of moderate difficulty. Designing and implementing (and explaining) the necessary scaling rules is never an easy job.

A3. The source language will require global (to a scope)

specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.P

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedT

Not fully met. Global arithmetic precision cannot be specified, and it is machine dependent (p. 204).

Precision can be specified for individual variables (pp. 44-48).

The addition of global arithmetic precision specification would not be much of a change to the language, but it would probably cost 3-6 man months per target machine to write the necessary library routines.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.F

Treated as exact quantitiesF
Range and step size determined at compile timeF
Scaling handled automaticallyF

Not met; CS-4 has no fixed point.

For the scope of changes necessary to add this feature, see the evaluation of A2.

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)F

Not met. New character sets cannot be defined. Only ASCII is provided (p. 100). There is no conversion between character sets (obviously, because there is only one).

It would be relatively easy to add the required capability.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P+

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeF
Upper subscript bound is fixed at scope entryT
Subscripts only integers or from an enumeration typeT
Subscripts will be from a contiguous rangeT

Partially met. The lower subscript bound is variable at run-time (p. 83) - a violation.

Modification to meet the requirement would be trivial.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.F

Alternative structures for records are possibleN/A
Discrimination condition may be any Boolean expressionN/A

Not met. Records do not have alternate structures (p. 87). A component may be of a discriminated union type, but strictly speaking, this is a different capability (see E6).

Meeting the requirement would cause a fairly major change, as CS-4 solves the problem in a different way.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.T

Automatically defined for any type (except...)T
Available for individual componentsT
(Assignment and reference via functions)F

Met. see pp. 25-26, 31, 39, 50, 60, 77, 89, 97, 104, 118, 163-164.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.P+

Partly met. There is an identity predicate (called COMPARE) for every type; it does a component by component comparison for arrays and records; and it works also for defined types (see pp. 31, 37, 50, 60, 77, 90, 97, 105, 118, 164-165).

However, the predicate does not allow comparing objects of possibly different type, except for UNION.

Change to meet the requirement would be relatively minor.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P+

Built-in for all numeric and enumeration typesT
Ordering can be inhibited when desiredF

Partly met. See pp. 34, 42, 52, 62-63, 80. There is no way to declare unordered types.

A very simple change would be needed to provide unordered enumeration types. The PRED and SUCC functions would clearly be undefined for such types.

B4. The built-in arithmetic operations will include:
addition, subtraction, multiplication, division (with a real
result), exponentiation, integer division (with integer or
fixed point arguments and remainder), and negation.P+

AdditionT
SubtractionT
MultiplicationT
Division with real resultT
ExponentiationT
Integer and fixed point division with remainderP
NegationT

Partially met (see pp. 67-75). Violation: Integer division has no
remainder (p. 72), and there is no modulus operator (pp. 41, 43).

A very easy addition.

B5. Arithmetic and assignment operations on data which are
within the range specifications of the program will never
truncate the most significant digits of a numeric quantity.
Truncation and rounding will always be on the least
significant digits and will never be implicit for integers
and fixed point numbers. Implicit rounding beyond the
specified precision will be allowed for floating point
numbers.T

Never from the left for data within rangeT
Never on the right for integer and fixed pointT
Implicit floating point rounding beyond precision allowedT
(Run time checks can be avoided)F

Met. See pp. 39, 50, 60, 67-75.

B6. The built-in Boolean operations will include "and",
"or", "not", and "xor". The operations "and" and "or" on
scalars will be evaluated in short circuit mode.T

Short-circuit andT
Short-circuit orT
NotT
XorT

Met (see pp. 32-33). In addition to the required operators, CS-4 also has NAND, NOR, EQV, and IMP.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

....F

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF

Not met. There are no scalar operations on arrays (p. 93), and assignment is legal only if source and sink are of the same type (p. 125).

Change should not be too hard, as component by component compare operations are already available.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

....P

No implicit conversionsF
Explicit between integer, fixed point, and floating pointP
Explicit between fixed point scale factorsF
(Explicit between integer and Boolean)F
(Explicit between integer and enumerated types)T
(Explicit between different enumerated types)P-

Partially met. Violations: The arithmetic operations all work with mixed mode operands (pp. 71ff) which can be interpreted as implicit type conversions. There are no fixed point conversions.

Required explicit conversions are discussed on pp. 40, 51, 56, and 106.

The required change to the language would be fairly minor, but many would argue that it is unnecessary and even undesirable.

B89. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.T

Implicit conversion between rangesT
Exception condition on integer and fixed point truncationU

Met. See pp. 39, 50, 60, 67-75.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.P

Sending and receiving of dataT
Sending and receiving of control informationF
Dynamic device assignmentF
User exception condition controlT
Installation independenceU
(Data formatting capability)T
(Reading and writing of bit strings)F

Partly met. (see Part II, pp. 1-20).

Violations: (1) No provision for sending and receiving control information as such. (2) No dynamic device assignment.

It is unknown from the documentation if the capability is installation-independent.

It would be an easy to moderate change to the language to include these missing features.

Requirement B11

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

....P+

Union	T
Intersection	T
Difference	F
Complement	T
Membership predicate	T
(Set inclusion)	T

Not fully met (see pp. 253-256). There is no difference operation; A-B can be written indirectly as NOT (A IMP B).

Trivial change necessary.

Requirement C1

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.T

Side effects must occur in left-to-right orderT
(Embedded assignments)F

Met (p. 28), although we wonder whether implementations will in fact be so strict.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.Pt

Few precedence levelsT
No user-defined precedence levelsT
Operands of an operation are obviousP

Partly met. There are only 7 precedence levels (unary + and - are of higher precedence than * or /) (see pp. 211-214). There are no user-defined precedence levels.

Violation: The operands of an operation are not always obvious, because A/B/C and A/B*C are permitted (p. 67).

It would be an easy change to require explicit parenthesizing in the above cases.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.Pt

Partly met. There is however an odd restriction on Boolean expression forms: I > 0 AND P is not permitted; it must be written (I > 0) AND P (pp. 34-35).

The change appears so easy that we wonder whether there is some hidden difficulty arising from a specialized parsing technique.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

....T

Met. (p. 173).

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.

....P

Parameter rules consistent in all contextsT
No special operations applicable only to parametersF

Partly met (see pp. 140-147, 150-151, 160-162, 170-172). The violation is that the operation of dynamically binding a procedure name to a procedure body is available only in the parameter context.

It is not a violation that initialization is allowed for (COPYO) procedure parameters (pp. 140-141), but not for parameters to a mode declaration (pp. 160-161), because COPYO parameters are not allowed in the latter case. Permitting fewer options in some cases is not an inconsistency.

To bring CS-4 into conformance with this requirement would be a major change. To do it right, a pointer mechanism should be added.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

....T

Actual and formal parameters will agree in typeT
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedT

Met (pp. 143-147).

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P

Act as constants (call by value plus)	F
Act as variables (call by reference)	T
Exception control	P
Procedure parameters	T
(Act as variables, but call by value)	T
(Act as variables, result parameter)	T

Partly met (see pp. 139-153).

Violations: There is no way to declare that a parameter is a constant within the procedure (p. 139-151). There is a way to handle exception conditions by a sort of implicit parameter which is a "signal-handling" procedure. When an exception condition occurs, a search is made through the dynamic control chain for a procedure which can handle this exception, so such procedures are in effect implicit parameters. They may also presumably be explicit. However, there is no special format parameter class for exception handling, so strictly speaking, CS-4 does not meet this part of the requirement. (p. 151-155). See also G7.

CS-4 also has formal parameter classes which the Tinman disallows: Call by value and/or result (copy in and copy out), with no restriction against change within the procedure.

Scope of necessary change: major. The procedure passing mechanism is one of a language's most vital organs.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.P

Above properties optionalF

Above properties are fixed at run timeT

Partly met. Formal parameter attributes are not optional in CS-4; a formal parameter must be declared with a full "mode - invocation" (pp. 139-140). In the discussion of open procedures (pp. 148-149), it is implied that the formal parameter attributes can be left "unresolved", but this apparently contradicts this syntax on p. 140.

Scope of change: Fairly major, if done right. One wants to leave unspecified just those parameter attributes not needed directly in the procedure; the operations to be applied to the parameters should be specified, and this is a whole new notion of "type".

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

....P-

Variable number of arguments possibleP-
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeT

Partly met. The number of actual parameters may vary if the matching is by keyword, but the number of formal parameters is fixed. That is, if the actual parameter is not given, then a user-specified default value is assigned to the formal parameter. There is no direct way to get a count of the number of actual parameters that were supplied on a particular call. This does not meet the needs of, for example, a print procedure, so this part of the requirement is only partially satisfied. (p. 145).

There is no restriction to the same type for optional parameters. (p. 143-145).

Scope of change: Moderate to difficult, depending on the flexibility desired.

D1. The user will have the ability to associate constant values of any type with identifiers.T

Met (pp. 15-16), although in a very awkward fashion. A range specification must be given for constants!

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesT
Consistent interpretation in program and dataU

At least partly met. There are literals for all built-in types (pp. 7-9). However, the input conversion description does not mention accuracy (Part II, p. 18), so it is unknown whether this part of the requirement is met.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P+

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesF

Almost completely met (pp. 18-19). An object of a user-defined type may have a default initial value (p. 166). This is technically a violation, but it seems necessary.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will

be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....P+

Numeric variable range specification mandatoryT
Fixed point variable step size specification mandatoryF
Range and step size specifications do not define a new typeT

Partly met. Range does not define a type, and the range specification is mandatory for numeric variable (integer: pp. 37-38; real: pp. 44-46; fraction: pp. 58-60).

However, there is no fixed point, ergo no step for fixed point.

The necessary modification is part of the work of adding fixed point -- see the evaluation of A2.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

....P

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataT

Partly met. There are no arbitrary restrictions on the structure of data (pp. 82-84, 94-95), but there is no way to associate a range with a variable of an enumeration (STATUS) type. (p. 76).

The necessary change is minor.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....F

Recursive and network structures providedF
Handles variable-value and structure-component connectionsF
Pointer property is an attribute of a typed variableF

Pointer property not for constants, affects only assignment	N/A
Pointer property mandatory for dynamic allocation	F
Allocation scope never wider than access scope	U
(Either the value or the pointer is modifiable)	F
(Pointer mechanism handles procedures and parameters)	F
(Remap and replace assignment have different syntaxes)	N/A
(Built-in dynamic variable creation)	T
(Variable equivalence classes are declarable)	F

CS-4 has no pointer mechanism as yet (p. 79-80 of Part III). Indeed, even in the section discussing a future pointer capability, the comment is made that recursive data structures are being considered as an alternative. This shows that the planning had not proceeded very far: recursive data structures alone cannot support all needed pointer capabilities.

If done right, addition of a pointer mechanism would be a major effort, and would effect such other vital features as parameter passing.

A rudimentary pointer mechanism (e.g., similar to that in PASCAL) could be patched onto the current definition with moderate effort.

E1. The user of the language will be able to define new data types and operations within programs.

....P

Partly met. New types can be defined, but new operators can not (pp. 157-172). New operations for the new types can be defined, but are accessible only via procedure calls.

Necessary changes are of moderate difficulty. The expression parser is rather strongly affected.

E2. The "use" of defined types will be indistinguishable from built-in types.

....T

Partly met (pp. 17, 170-172). Components of data objects defined by the user are not accessible directly (see pp. 117, 163). The only infix operators usable with a new type are = and ~= (p. 169), but the same is true of some built-in types (e.g., STATUS). Assignment can be defined for a new type (pp. 163-164).

The necessary change is of medium difficulty. It interlocks closely with the addition of user-defined operators (E1).

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

....T

Met. (pp. 2, 14-16).

E4. The user will be able, within the source language, to extend existing operators to new data types.

....F

Not met. Only the = and ~= operators can be extended to new types (p. 164).

Scope of change: Moderate. It is closely connected with provision for user-defined operators (E1).

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.P+

ConstructionT
SelectionP
PredicatesP+
Type conversionsT
Operations and data can be defined togetherT

Construction is a special case of the CS-4 concept of mode-invocation.

Selection is accomplished by means of the standard dot-qualification. For example, if X is a variable of a defined type (mode in CS-4 terminology) and I is a "representational entity" of that type, then X.I refers to the specific instance of I contained within the "value" of X. A minor deficiency here is that all "representational entities" of a defined type are accessible in this manner. It would appear that the person who defines a type might want to hide or protect some of the "representational entities".

Predicates and type conversions can be defined by means of procedures. Predicates are not definable as special operators (e.g., infix), however (see E1).

The ability to declare a "representational entity" to be inaccessible would require a fairly minor change in the language. To allow special operators is much more complicated (see E1).

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.T

EnumerationT
Cartesian products (records)T
Discriminated unionT
Powerset of an enumeration typeT

Met. Enumeration is called STATUS (pp. 76-82), Cartesian products are called STRUCTURES (pp. 94-99), discriminated union is called UNION (pp. 113-121), and powerset is called SET (pp. 253-256).

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.T

Met. The only unions are discriminated (pp. 113-121), and there is no way to define a type (mode in CS-4 terminology) as a subset of another.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.T

InitializationT
FinalizationT
Allocation actionsT
Deallocation actionsT

Met. Initialization and allocation operations can be done in a user-defined INIT procedure (pp. 165-167). Finalization and deallocation can be done in a TERM procedure (p. 167).

Requirement F1

F1. The language will allow the user to distinguish between scope of allocation and scope of access.P

Partly met. Allocation and access scopes can be different, but only if the allocation scope is the whole program, similar to ALGOL 60 own (n. 20), or if the data object is shared. In the following block structure, it is not possible to have an object whose allocation scope is block B2 and whose access scope is B3.

```
B1: BEGIN
    P2: RBEGIN
        B3: RBEGIN
            :
            :
        END; END; FND
```

The change should be relatively easy. The AUTOMATIC storage class declaration would simply have to carry an optional parameter, giving the block name of the allocation scope block.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.T

Allowable operations can be limitedT
Access can be limited where usedT
External declarations need not all have the same scopeT
Naming conflicts can be avoided (renaming)T

Met. Limiting of allowable operations is discussed on pp. 157-172. Limiting access to external entities and renaming are discussed on pp. 174-176.

F3. The scope of identifiers will be wholly determined at compile time.T

Met (pp. 14, 126-127, 140).

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.0

Unknown. It is implied (pp. 174-177) that the compiler will have access to entities external to the program being compiled, but it is not stated that these entities will be in libraries, and no tools for building such libraries are defined.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.?

Program component libraries accessible at compile time0
Libraries can contain foreign language routines0
Interface requirements checkable at compile time0

Unknown. See F4.

F6. Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.0

Libraries and compools will be indistinguishable0
Immediately accessible sublibraries at any level0

Unknown. See F4.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

Not met. The CS-4 Operating System Interface provides file handling and parallel processing, but these are covered by requirements 21c and G7, not this requirement. There are no provisions for machine independent interfaces to specific peripheral equipment or special hardware.

Meeting the requirement would necessitate a major design and implementation effort, and indeed, we are not sure how one should go about it.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

....P+

Sequential executionT
Conditional executionT
IterationT
RecursionF
(Pseudo) parallel processingT
Exception handlingT
Asynchronous interrupt handlingD
Control structures from a small set of simple primitivesF

Not fully met: CS-4 has no recursive procedures as yet (pp. 139-140). CS-4 does have interrupt handling, by means of SIGNAL-handling procedures (pp. 152-155).

The set of primitive control structures is larger and more complicated than necessary. EXIT and ABORT TO are simply euphemisms for GOTO, hence are superfluous. Worse, for parallel processing there are nine scheduling procedures, four synchronizing ones, six parameters describing priority, and eight process states. (Part II, pp. 21-34). This hardly seems a small set, or a primitive one.

Scope of change: Major. Recursive procedures are never easy to add, and they effect much else, particularly in implementation. In our view, no one has yet designed a good set of parallel processing capabilities for a language -- the problem must be hard.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

....P

Partly met. The CS-4 GOTO can transfer control out of a scope level, although not into a scope or out of a procedure (pp. 135-136).

We regard the ABORT TO statement as only a syntactically camouflaged GOTO, and it leads out of an exception handling procedure (p. 137).

Scope of change: Relatively minor.

Requirement 63

63. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.T

Based on Boolean expressionT
Based on type from discriminated unionT
Based on computed choice among labeled alternativesT
All alternative must be accounted forT
Simple mechanisms will be supplied for common casesT

"Met (pp. 122-131). Full discrimination for an IF is provided by the closing FI. The IF is a simple special-case mechanism, and it need not have an ELSE.

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).P+

Termination can occur anywhere in the loopT
Multiple terminating predicates are possibleT
Entry permitted only at the loop headT
Simple cases are clear and efficientP+
Control variable is local to the loopT
Control value is efficiently available after terminationF

Partly met. The CS-4 iteration statement (REPEAT, optionally preceded by WHILE and/or FOR) is fairly clean and efficient, and meets the requirement except:

(1) There is no clean mechanism for the very common "n and a half" times loop. An EXIT, tantamount to a GOTO, must be used:

```
L: REPEAT
    {get data};
    IF {exhausted}
        THEN EXIT L
        ELSE {process data} FI;
    END
```

S.J. Dahl's syntax avoids the EXIT:

```
loop
    {get data}
    while {not exhausted} : {process data}
repeat
```

(2) CS-4 has no means of making the value of the control variable accessible after termination of the loop, except by assigning it to a variable not local to the loop on each iteration, and this is hardly the most efficient way. (pp. 132-134).

Difficulty of change: easy to moderate.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

....F

No recursive procedures within recursive proceduresF
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)F

Not met. CS-4 has no recursive procedures -- a major lack.
(pp. 139-140).

Scope of change: major.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

....P

Able to create and terminate parallel processesT
Process can gain exclusive use of resourcesT
No parallel routines within recursive routinesT
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF
(Access rules are enforced)T

Partly met.

A parallel procedure in CS-4 is not identified as such, and may presumably also be called in-line, even recursively. There is no way to specify the maximum permissible number of simultaneous instance of a parallel process.

It is not clear when the parameters of a parallel process are bound; presumably when the process is scheduled. It is evidently not possible to define a parallel procedure within the body of a recursive or a parallel procedure (Part II, p. 24).

The CS-4 parallel processing capability, in short, while in our opinion far too complicated, satisfies the G6 requirement except that the maximum number of simultaneous activations is not declarable.

Only minor change is needed.

The capability is in direct conflict, however, with requirement G1; an operating system is required. For example, deadline scheduling implies that the CS-4 scheduling mechanism has control of the entire machine. The capabilities are even described under the heading "Operating System Interface".

Conflicts, Scope

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.P+

Program can get control for any exceptionT
Parameters can be passedP
Can get out of any level of a nest of controlT
Can handle the exception at any level of controlT

Partly met.

Exceptions caused by "checking directives" (e.g., zero divide) carry no arguments (p. 175).

To get out of any arbitrary nest of control, an "ABORT TO Label" statement is used (p. 137). This is in effect a GOTO out of a procedure, but an unavoidable one. The advantage of using ABORT TO rather than GOTO is not clear.

Relatively minor change necessary.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

....P+

Priority specificationT
Synchronization via wait/enable operationsT
Wait for end of real time intervalT
wait for end of simulated time intervalF
Wait for hardware interruptT
(Can enable and disable interrupts)T

Not fully met. There is apparently no concept of simulated time in CS-4. The TIME-WAIT procedure (Part II, p. 67) evidently causes a wait in real-time only. See also Part II, pp. 21-32, 63-67.

"minor change necessary.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....P-

Free format with statement terminatorT
Mnemonic identifiers possibleT
Based on conventional formsP
Simple grammarF
No special case notationsF
No abbreviations of identifiers or keywordsT
Unambiguous grammarT

Only partly met.

The syntax of CS-4 is not one of its better features. It does meet part of the requirement: It is free format with a statement terminator, it allows identifiers to be up to 32 characters long, the grammar is (to our knowledge) formally unambiguous, and there is almost no abbreviation of words (exceptions are & for AND, | for OR, and ~ for NOT).

However, the language is not easy to read. ATTR is used to introduce attributes of either formal parameters or of the whole procedure, so that parenthesis counting is often necessary to sort things out. The array declaration is very hard to read; for example:

```
ARRAY(1 THRU 2, INTEGER (RANGE:1 THRU 10) [CEIL(X)] THRU 20,  
      REAL (RANGE: 0 THRU 10000, PRECISION: 10))
```

Not until ']' is reached is it clear that the INTEGER is a call on a conversion routine, not the type of the array members. It would have been much better to separate the types of the subscripts and the numbers.

Similarly, the mandatory range specifications on most type conversion calls make them often all but unrecognizable; see for example the conversion on page 149 of the CS-4 Manual -- it takes 11 lines! The worst example of such wordy, over-paternalistic syntax is the constant declaration; this example appears on page 16:

```
CONSTANT TWO IS INTEGER (RANGE: 2 THRU 2) [2]
```

Why not

```
CONSTANT TWO = 2
```

or, if necessary,

INTEGER CONSTANT TWO = 2

Hence we believe that while the CS-4 syntax may be formally unambiguous, it can be very hard for a human to read.

Moreover, the rule for blanks is very dangerous. Spaces may be inserted or omitted at will except where ambiguity arises. For example, GOTOL1 is presumably acceptable unless there is also a procedure named GOTOL1. Among other things, this makes it dangerous to add new identifiers to an existing program.

(The manual is not wholly clear on this matter of spaces. Using a different interpretation of the phrase "valid token" on page 5, one could say that GOTOL1 is not allowed, but in that case neither is GOTO -- only GO TO -- which has its own problems).

Nor can the syntax be regarded as simple: There are some 255 syntax definitions. The initial definition of PASCAL, by contrast, has less than 100; ALGOL 60 (Revised Report), less than 125. It is not clear whether the extra complexity of CS-4 is in the language or the syntax definitions; either way, there is a problem. The CS-4 syntax is also complex enough to still contain rather basic errors; for example, it allows a procedure to be a component of an array or structure. (see pp. 29, 94). This is clearly a mistake, because there is no way to reference such a procedure.

The syntax is also not wholly consistent. Procedure parameters are enclosed in parentheses except for object-construction procedures, where square brackets are used. The CEIL, FLOOR, and ROUND conversion procedures are syntactically different from other conversion procedures.

Scope of change: Major. A redesign of many of the most complex parts of the syntax is necessary, and a clean syntax is not easy to design.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.1

Met. The language includes no macro facilities, and no tools for changing or adding to the syntax.

H3. The syntax of source language programs will be convertible from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.F

This requirement is not met. CS-4 uses 94 printing characters, in addition to the new-line sequence and the blank (p. 5).

This requires only a relatively minor change. The only characters not in the ASCII subset are the lower case letters, {, }, and ^.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.P+

Break character existsP
(Literals are self-identifying as to type)T
(Bit-string literals for any type)F

This requirement is partly met (pp. 6-10). There is a break character (underscore) for identifiers, but not for literals. Literals for numbers and character strings are provided.

Any necessary change would be minor.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.F

This requirement is not met. The new-line sequence is treated as a space, which cannot appear in string and status literals, but which apparently is ignored in identifiers if no ambiguity arises (pp. 9-10).

Any necessary change would be minor.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where

an identifier can be used.P

This requirement is partly met. In total, there are 267 key words, of which 43 are fully reserved. This seems more than a few. (pp. 261-263)

To reduce the number of keywords would be a fairly difficult change, as a syntax redesign would be required.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.T

Uniform comment conventionT
Look different from codeT
Bracketed by one or two charactersT
Can contain any charactersT
Can appear anywhere reasonableT
Terminated by the end of the lineT
Compatible with automatic reformattingT

This requirement is fully met. (pp. 9-10)

H8. The language will not permit unmatched parentheses of any kind.T

This requirement is fully met; see the syntax description (pp. 209-232).

H9. There will be a uniform referent notation.F

This requirement is partly met; see the syntax description (pp. 209-238). Calls on object construction procedures use square brackets; all others use parentheses.

The scope of any required change is minor.

H10. No language defined symbols appearing in the same context will have essentially different meanings.T

This requirement is fully met. Verification requires a study of the entire language manual.

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

....P

This requirement is partly met. The precision for floating point arithmetic is not specifiable, and it is implementation dependent (pp. 71, 204)

Only a fairly minor change would be required to specify floating point precision.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

....Pt

Defaults specified for don't care casesP
Programmer can override the defaultsT

This requirement is partly met. The compiler supplies defaults for standard procedures (ASSIGN, COMPARE, INIT, and TERM), for user-defined types, for procedure calls (CLOSED), for storage class (AUTOMATIC), and for checking categories (ENABLED); these can be overridden (p. 259). Only for MSTRUCTURES (pp. 189-193) must data representation be specified. However, nothing is said about reentrant code.

A minor change is needed.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

....T

This requirement is met. While there are only a few built-in compile-time "variables" describing the machine configuration (e.g., MAX_MACHINE_INTEGER_VALUE, p. 204), it is possible to reference externally defined variables, which may have initial values (pp. 176-177).

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.F

This requirement is not met; CS-4 has no such capability.

The change would be of small to moderate difficulty, depending on the degree of integration of the new capability with existing ones.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.F

This requirement is not met. The built-in features make CS-4 probably twice as complex as PASCAL, for example.

To restructure CS-4 as a small kernel, with most currently built-in features provided as extension, would be a major effort, and might even change the language.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.F

This requirement is not met.

To change CS-4 to meet it would be fairly easy, but non-trivial.

Requirement I7

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.T

This requirement is met, except that it is unknown whether the translator will give a warning if object machine capabilities are exceeded.

Requirement J1

J1. The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

No efficiency cost for unused features
Efficient code can be produced for all features

This requirement is partly met. We think that efficient code for calling exception-handling procedures will not be possible -- it is not clear why the pertinent procedure is determined by a dynamic call-chain search. We also suspect that providing for ABORT TO statements will introduce overhead not otherwise necessary, particularly when recursive procedures are added. We note that the manual does not define whether COPYC parameters (p. 144) are set on an ABORT TO exit. The only optimization directive is NORECALL (pp. 148, 150); certainly others will be needed for really efficient code (e.g., specification of recursiveness, and of the maximum depth of recursion).

It is hard to pin down the scope of any necessary change in the absence of existing translators, but it probably is not minor.

J2. Any optimizations performed by the translator will not change the effect of the program.

....II

Unknown. This is a translator, not a language, requirement, and moreover the meaning of "change the effect" is itself undefined. In other words, the requirement cannot be evaluated for any language, even if it were restated with reasonable precision.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

....I

This requirement is fully met (nr. 193-101).

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will

be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

.....P-

Encapsulated specification of representation possibleP
Space/time tradeoff can be specifiedF

This requirement is partly met. The capability is provided only for records (not, for example, for arrays as in ALISS) (p. 140-123). Furthermore, only certain things about mapping can be specified; it takes a full left-hand-side function capability to handle all cases. There is no way to specify various densities of packing for structures.

A moderate change is required, depending on the power of the final mechanism.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

....T

Open/closed properties can be specifiedT
Open and closed versions have the same semanticsT

This requirement is fully met (p. 148).

Extraneous Features

We recommend that the following features of CS-4, not required by the Tinman, be kept:

- * COMPLEX type and operations.
- * STRING type (assuming that the Tinman requires only a character type, not a character string type).
- * Trait inquiry procedures.
- * Various explicit conversions: String to Boolean, status to or from integer, string to integer, string to status.
- * Real relational procedures, which perform the comparison to a specified precision.
- * Various procedures: ABS, SEN, SQRT, logarithmic, exponential, and trigonometric procedures, SUCC and PRED.
- * Mixed-mode arithmetic.
- * COPYI, COPYO parameter classes (call by value and result, respectively).

We recommend that the following features of CS-4, not required by the Tinman, be deleted:

- * VECTOR and MATRIX types and operations.
- * Cross sections of arrays.
- * Keyword parameters.
- * Fraction type.
- * Authorities for use of features.
- * Extra Boolean operators: NAND, NOR, TNE, and EQV.
- * NAME parameters.

The features recommended for deletion provide only convenience, not additional power (e.g., a user can define his own MATRIX type, albeit without being able to use infix operators for it). In our judgment, the extra convenience is not worth the extra complexity, compiler size, etc., in the cases noted.

COMPLEX type is an exception; although it too could be defined by a user, it is a numeric type and the ability to use infix operators is overriding. If, however, CS-4 is extended to permit the extension of built-in operators to new data types, then COMPLEX type could be reasonably excluded from the base language.

Summary

CS-4 is a curious language. In many important respects, it conforms to the spirit of the Tinman, and contains many features which have recently become fashionable. In particular, it has:

- * Strong type-checking, with few implicit conversions.
- * Powerful data structuring tools, with full safety.
- * Ability to define new abstract types (although not new infix operators).
- * Support for parallel processing and exception control.
- * User control over precision and range of variable values.
- * Complexity occurring mostly at compile time (i.e., efficient code should be possible in most cases).
- * Reasonably structured control mechanisms (although it is not outstanding in this area).

It also includes most of the features necessary for systems programming. For example:

- * Variable initialization.
- * Attention to separate compilation.
- * User control over mapping of (some) structures into physical store.
- * Access to assembly language code.

In addition, the language has fairly good basic consistency. For example:

- * A function can return a structure as its result.
- * Expressions are acceptable in place of variables or constants in most contexts.
- * Arbitrary restrictions are rare.

On the other side, CS-4 would require major change to satisfy the Tinman in the following important areas:

- * Procedure parameter classes; variable number of parameters.

- * Generic procedures.
- * Pointers (CS-4 has none!).
- * Recursive procedures (CS-4 has none).
- * Machine independent interfaces to hardware components.

Of these, recursive procedures and pointers are the most important. They are basic features which have long been in some languages, and which affect (or should affect) many other vital features. It is not so clear that procedure parameters, generic procedures, and machine independent interfaces to hardware can be provided in any language in a way consistent with the spirit of the Tinman, yet efficient and clean.

Moreover, many of the more modern features have been included in what we can only regard as a pedestrian, often clumsy, way. This is clearly shown by the overall awkwardness of the CS-4 syntax (see the comments on requirement H1), the complexity of the parallel processing features, and the wordiness of declarations of new data types (even though new operators cannot yet be declared). CS-4 is also a large language, far from simple, as is clearly shown by the number of syntax definitions -- and the syntax appears to be rather carefully written. Certain features seem unnecessary; for example, the fraction type (floating point provides as much or more, in most cases), the six calling modes for procedure parameters, the system of authorities to permit a programmer to use certain language features (the compiler could instead easily print what he did use, for perusal by his manager if necessary). Another important missing capability is a system of assertions to permit the compiler to safely avoid run-time checking code.

It seems clear that if one starts with a big, rather awkward language like CS-4 and tries to modify it to meet the Tinman requirements (or any other, perhaps superior, requirements), the result will be an even bigger, clumsier, language. The requirements for simplicity, code efficiency, etc., may simply not be satisfiable given the CS-4 starting point. CS-4 does have the advantage that it doesn't really exist yet, so there are no users to insist that certain unwanted features be kept; perhaps an effort to first pare down CS-4 to a minimum, clean it up, then build it up to the requirements, might be successful. Starting with a simpler, cleaner language in the first place seems cheaper, and far less risky.

A COMPARISON OF
EUCLID
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language EUCLID to the Tinman Language Requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1974, Section IV). For the purposes of this comparison, EUCLID is considered to be defined by:

Report on the Programming Language EUCLID
F. W. Lampson, J. J. Horning, K. D. Landon,
J. G. Mitchell, and C. J. Popk
July, 1976

Tinman contains 78 language requirements. This report compares EUCLID to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols F, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of EUCLID which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring it into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.T

This requirement is fully met. All identifiers must be declared before use (p. 7) and part of the declaration for data is the specification of their types.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.P

Integer	T
Floating Point	F
Fixed Point	F
Boolean	T
Character String	T
Arrays	T
Records	T

EUCLID has no floating point or fixed point data types. They are unnecessary for the purpose of the language. The other types are all available (pp. 17-22).

The addition of these two missing types is a relatively simple task. Fixed point gives the most difficulty, in particular in defining acceptable scaling rules and in added complexity in the code generator, but neither of these is particularly difficult.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.F

Global arithmetic precision specification mandatoryN/A
Individual variable precision specification permittedN/A

EUCLID does not have a floating point type at all.

Adding such a type to the language, with the capabilities specified in this requirement, would be a relatively simple task.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.F

Treated as exact quantitiesN/A
Range and step size determined at compile timeN/A
Scaling handled automaticallyN/A

EUCLID does not have a fixed point type at all.

Adding such a type to the language, with the capabilities specified in this requirement, would be at worst of moderate difficulty. There would be a moderate impact on the code generator and the definition of acceptable scaling rules is bothersome. (In general, it is not practical to treat fixed point quantities as exact.)

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)N/A

EUCLID does not address the question of multiple character sets, nor is there any need to because the language does not support any input/output; the native character set of the machine is sufficient.

It would be relatively easy to add the required capability.

EUCLID
Requirement A6

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.
T

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryT
Subscripts only integers or from an enumeration typeT
Subscripts will be from a contiguous rangeT

The EUCLID definition of array type satisfies most of the features of this requirement in spirit. EUCLID allows only one-dimensional arrays, so the number of dimensions is vacuously fixed at compile time. Subscripts are from an index type, which is a contiguous range of either the integers or an enumeration type. The lower subscript bound is fixed at scope entry, not compile time.

In addition, the type of an array which is allocated at run-time can be a record with alternative structures, with the particular structure specified when the array is allocated. By the EUCLID use of the word type, the type of such an array is known at compile time, but this might not be in the spirit of this requirement.

To add multi-dimensional arrays to the language is trivial, as it is to require that the lower bound be fixed at compile time. Nor would it be much harder to rigorously require that the type of an array be known at compile time, if that is truly wanted, but this would spoil the elegant regularity of EUCLID's existing data structuring mechanisms.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.
T

Alternative structures for records are possibleT
Discrimination condition may be any Boolean expressionU

This requirement appears to be fully satisfied by EUCLID. However, although the case statement has a special syntax for discriminating alternative structures, the language itself makes no statement about, nor is any example given of, discriminating by means of an arbitrary Boolean expression.

EUCLID
Requirement A7

4

If any change is necessary, it would be a very minor one.

R1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.P+

~~Automatically defined for any type (except...)~~P+
~~Available for individual components~~T
~~(Assignment and reference via functions)~~F

In general, a value of any type can be assigned to a variable of that same type, subject to the FUCLID definition of identity of types. In addition, variables which have alternative record structures can be assigned a value which is one of the alternatives. The opposite assignment is not valid; the fields of the record must be assigned individually in this case. (p. 43) However, if the value is of a module type -- a generalization of the record type which may or may not include storage management -- the definition of the type must explicitly specify that assignment is an allowable operation (p. 22).

R2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.P+

The equality operator is available for any two data objects of the same type, but -- as for assignment -- if that type is a module type, the definition of the type must explicitly specify that comparison is an allowable operation (see the comments on requirement R1). Comparison of objects of different types (giving a false result) is not permitted (p. 41).

The change needed to allow comparison of data of different types would be minor.

R3. Relational operations will be automatically refined for numeric data and all types defined by enumeration.P

Built-in for all numeric and enumeration typesT
Ordering can be inhibited when desiredF

EUCLID
Requirement B3

6

The six relational operation are available for both numeric (integer) and enumerated types (p. 41). It is not possible to inhibit the ordering of enumerated types, however.

To allow inhibiting the ordering on enumerated types would be a trivial change.

B4. The built-in arithmetic operations will include:
addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

Addition	T
Subtraction	T
Multiplication	I
Division with real result	F
Exponentiation	F
Integer and fixed point division with remainder	I
Negation	I

Exponentiation and division with a real result are not defined in EUCLID, the latter because the language does not support floating point at all. Because fixed point is not supported at all, the "integer" division operator (division with truncation) accepts only integer operands. In addition the language has a remaindering operator, mod.

Adding the missing operators to the language definition is a simple task, although the proper definition of exponentiation for fixed point operands is open to some discussion. There is a moderate to heavy impact on the code generator, particularly if two new numeric types (fixed point and floating point) are added at the same time.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

Never from the left for data within range	F
Never on the right for integer and fixed point	F

Implicit floating point rounding beyond precision allowedN/A
(Run time checks can be avoided)

The language definition requires that the compiler verify at compile time that overflow will not occur or that it generate a legality assertion to be processed by the program verifier (p. 40). Integer data is assumed throughout the report to be exact, although no explicit statement to this effect is made, therefore precluding truncation on the right. (The result of integer operations is required to be integer -- p. 40.) The language has no fixed point type, so the question of truncation on the right does not arise. Nor does the question of rounding of floating point data arise, because that type is not supported either.

If the missing data types are added to the language, it will be a simple problem to require them to satisfy this requirement. It would be a minor change to require that code be generated instead of a legality assertion in order to check for overflow at run time. If such a feature is added to the language, it should be possible to disable the run time checks.

R6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P

Short-circuit andF+
Short-circuit orF+
NotT
XorF

The EUCLID report does not require that short-circuit mode of execution be used for and and or, but it contains the following statement (p. 41):

The right operand of and need not be legal if the left operand is False; the right operand of or need not be legal if the left operand is True.

The xor operator is restricted to powersets.

There would be almost no impact to explicitly require short-circuit evaluation of and and or, because it is the easiest way to implement the above requirement. Extending xor to Boolean data is simple.

EUCLID
Requirement R7

R7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

....F

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF

Scalar operations on arrays are not defined in EUCLID and the assignment operation is more restricted than that envisioned here (see the comments on requirement R1). Specifically, assignment between arrays and records are permitted if the assignment operands are of the same type, which means that their type definitions must be identical after certain substitutions are made. This excludes assignment between arrays and records which have the same logical structure but different physical structure.

The required capability is presumably similar to the ABOVE CORRESPONDING of COBOL. This is a moderately expensive feature, although its difficulties are well-understood.

R8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

....P

No implicit conversionsT
Explicit between integer, fixed point, and floating pointN/A
Explicit between fixed point scale factorsN/A
(Explicit between integer and Boolean)F
(Explicit between integer and enumerated types)T
(Explicit between different enumerated types)F

EUCLID lacks the missing conversion operations because it has no need for them. Fixed point and floating point types are not supported. Conversion between the character string form of a number and its internal form is not needed because the language has no input/output.

If the missing data types are added to the language, the numeric conversion operations can be added at the same time at modest cost. If input/output is defined, the numeric-character conversion is almost free because it is embedded in the input/output package.

EUCLID
Requirement R9

R9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.F

Implicit conversion between rangesT
Exception condition on integer and fixed point truncationF

The EUCLID report merely specifies that a program is illegal if a value is outside of the range of a variable to which it is assigned; it does not require that any code be generated to check for such potential truncations (p. 43). It also requires that no overflow occur during the evaluation of numeric expressions (see the comments on requirement R5).

It is a simple matter to add this requirement to the language definition and is consonant with the verifiability and legality assertion philosophy of the language. The possibility of truncation should be handled in a manner similar to the possibility of overflow (see the comments on requirement R5).

R10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.F

Sending and receiving of dataF
Sending and receiving of control informationF
Dynamic device assignmentF
User exception condition controlF
Installation independenceF
(Data formatting capability)F
(Reading and writing of bit strings)F

EUCLID has no input/output capability at all.

To add an input/output capability to EUCLID, particularly of the type envisioned by this requirement, is a major task. The only simplifying aspect of that task in this case is that the designer can start from scratch, rather than having to try to embed new I/O capabilities into existing features.

R11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.P+

Union	I
Intersection	I
Difference	T
Complement	T
Membership predicate	I
(Set inclusion)	I

Of the requested operations, only the set complement operator is not available in EUCLID (pp. 39-41).

It would be a minor problem to add the set complement operator to the language. The reserved word not could be used, thus not increasing the reserved word list, but other alternatives are available.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.F

Side effects must occur in left-to-right orderF
(Embedded assignments)F

Because of the simplicity of EUCLID, the only possibility of side effects within an expression would be in function references. The language designers have attempted to rule out side effects entirely by severely restricting the types of formal parameters which a function can have and allowing a function to communicate with the rest of a program only through its parameters (p. 10). Unfortunately, a function can call a procedure, and the execution of the procedure can change the value of a "global" variable -- a side effect to the function call. Thus EUCLID does not satisfy this requirement.

It would be a fairly easy task to initially design a compiler to satisfy this requirement, although we consider the requirement to be questionable. The heaviest impact would be in the code optimizer, which would be restricted in the code motion it would be permitted. Since there are few EUCLID compilers in existence (perhaps none), the cost of such a change to the language specification is minimal. It is probably better, however, to change the specification to rule out side effects entirely -- i.e., to forbid procedure statements within function bodies.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.P

Few precedence levelsT
No user-defined precedence levelsT
Operands of an operation are obviousP

EUCLID has six precedence levels; a fairly standard number which presumably satisfies this requirement (p. 30). There is no facility in the language for changing the defined precedence level. The usual convention of left-to-right interpretation of operators of equal precedence holds (p. 39), so that the constructs A div B div C and A div B * C (A/B/C and A/E*C respectively) are possible, as is A or P xor C for powersets.

To change the language to require explicit parentheses in the cases mentioned above, and other similar cases, costs little, particularly if the compiler's expression analyzer is syntax driven. The changes would

only be in the description of the syntax of expressions (which would become quite a bit more complicated through the invention of a number of new non-terminals), but would have no effect on any other part of the compiler.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.T

This requirement is fully met.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.T

This requirement is fully met (p. 32).

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.P

Parameter rules consistent in all contextsF
No special operations applicable only to parametersT

only three object in EUCLID may be parameterized: procedures, functions, and types. The permissible formal parameters to functions and types are more restricted than those for procedures, in an effort to eliminate side effects. (See the comments on requirement C1.) Formal parameters are considered to be variables declared within the scope which is the procedure, function, or type body. As such they have no semantics not held by any variables declared within some scope smaller than the entire program.

It would be a simple change to make the parameter rules for functions and types the same as those for procedures and would make any compiler simpler. It is doubtful that this is desirable; it would be better to change the requirement to agree with the EUCLID definition.

66. Formal and actual parameters will always agree in type.
The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.1

Actual and formal parameters will agree in type1
Rank of parameter arrays is fixed at compile time1
Parameter array size and subscript range can be passed1

This requirement is fully satisfied. Actual and formal parameters must be assignment-compatible if the formal parameter is treated as a constant and they must be of the same type otherwise (pp. 43-44). An exception to the latter rule is that a formal parameter can be parameterized and the corresponding actual parameter must then be a specific instance of the formal (pp. 51-52). This implements, among other things, deferring the fixing of the specified array properties until routine invocation. The properties can be passed either explicitly or implicitly.

67. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P-

Act as constants (call by value plus)1
Act as variables (call by reference)1
Exception control1
Procedure parametersF
(Act as variables, but call by value)F
(Act as variables, result parameter)F

EUCLID permits only two classes of parameters: Those which act as constants within the routine and the classical call-by-value. There is

no class for exception control because the subject is not addressed at all by the language. Procedure parameters have probably not been considered because they are unnecessary within the design considerations of the language.

To add the missing parameter classes to the language would be a fairly major change, but not too difficult. The exception control parameter would first require that the entire question of exception handling be addressed. Then, even if they are implemented by such a simple device as a label parameter class, additional changes would be required because EUCLID has no labels at present. The implementation of procedure parameters presents no particular difficulty.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

....F

Above properties optionalF
Above properties are fixed at run timeN/A

EUCLID has no generic procedure capability.

The addition of such a capability would be a fairly major change, but a well-understood one.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

....F

Variable number of arguments possibleF
All but a constant number of arguments have the same typeN/A
Number of arguments in each call is fixed at compile timeN/A

All parameterized entities in EUCLID (procedures, functions, and types) have a fixed number of arguments.

To add such a capability to the language would be a significant change. Not the least problem would be to find an acceptable syntax for

the routine declarations. Since EUCLID permits procedures to be both open and closed, and presumably these options would be available to routines with a variable number of parameters, a capability very close to a MACRO expansion would have to be added to compilers. This capability would be used for the closed case and could be used for the open case. Space considerations would probably require the definition of an elaborate linkage mechanism for the open case, however.

D1. The user will have the ability to associate constant values of any type with identifiers.T

This requirement is fully met. EUCLID has a constant declaration capability, and the constant can be of any built-in type, including the structure types. (p. 32)

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesP+
Consistent interpretation in program and dataF

EUCLID has literals for all the "simple" built-in types, including powersets, but not for the structured types (arrays and records). The question of consistency of interpretation between program and data does not arise because EUCLID has no input/output capability.

If an I/O capability is added to EUCLID it will be a simple further addition to require consistency between programs and data but, as the Tinman infers, this can cause amazingly expensive problems in implementation.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.T

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesT

This requirement is fully satisfied (p. 32). The language definition does not specify any default initial values, but it does not forbid any compiler to implement them.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....P

Numeric variable range specification mandatoryF
Fixed point variable step size specification mandatoryN/A
Range and step size specifications do not define a new typeI

EUCLID permits integer variables (the only numeric types supported by the language) to be specified as one of two integer subtypes, in which case their range becomes implementation dependent (p. 18). (Ranges of integers are also possible.) All integer variables are considered to be of the same type, regardless of their range specifications.

It is a trivial change to require that ranges be specified in all cases. If fixed point type is added to the language, adding the features of this requirement to the properties of the type definitions is equally trivial.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a continuous subsequence of any enumeration type.

....T

Ranges of an enumeration type are allowedI
No arbitrary restrictions on the structure of dataI

This requirement is fully satisfied. In particular, any structure may contain any type of structure as a component (pp. 19-23).

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....P

Recursive and network structures provided	T
Handles variable-value and structure-component connections	T
Pointer property is an attribute of a typed variable	F
Pointer property not for constants, affects only assignment	T
Pointer property mandatory for dynamic allocation	T
Allocation scope never wider than access scope	F
(Either the value or the pointer is modifiable)	T
(Pointer mechanism handles procedures and parameters)	F
(Remap and replace assignment have different syntaxes)	T
(Built-in dynamic variable creation)	T
(Variable equivalence classes are declarable)	F

EUCLID has a pointer mechanism with the following salient features:

- * Pointers point only to dynamically created variables, and they provide the only mechanism for referencing dynamically created variables.
- * Part of the declaration of a pointer is specification of the collection to which it may point. A collection is a group (number undetermined) of variables of the same type. The type may be parameterized, but the parameters are either fixed at allocation time or the pointer points to a variable of a discriminated union type. Thus pointers are as type-safe as any other variable.
- * The notation p^* is used to dereference the pointer p .
- * The value of a pointer to a collection can be assigned to another pointer to that same collection. Thus it is possible for a pointer to point to a variable which has been destroyed.

It is impossible to estimate how much of a change is required to modify the EUCLID pointer mechanism to fulfill this requirement, because the intent of the requirement is too unclear.

E1. The user of the language will be able to define new data types and operations within programs.P

EUCLID possesses the generally accepted data-structuring capability, with no arbitrary restrictions (pp. 19-22). In addition, it has the module type, a mechanism for defining a new type and any operations peculiar to that type (pp. 22-23). The defined operations can only be procedures and functions, however, not infix operators.

The changes necessary to permit the definition of infix operators on a user-defined type would be significant. It would be necessary to change the syntax of the module definition, but the required change would not be too radical. Major changes in the lexical analyzer and syntax analyzer would be needed, however.

E2. The "use" of defined types will be indistinguishable from built-in types.T

There are no syntactic differences between built-in types and user-defined types. In particular, functions may return values of any type (n. 54).

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.P+

Basically this requirement is satisfied by EUCLID: All identifiers must be declared. However, scattered throughout the report are a number of entities referred to as "standard" (e.g., string type and the functions abs and odd). Apparently these are intended to be standard extensions to a base definition, but in such a case it is difficult to distinguish them from part of the base language.

It would be a trivial change to satisfy this requirement by the fiat of declaring the "standard" entities to be part of the language.

E4. The user will be able, within the source language, to

extend existing operators to new data types.

....F

The only operations permitted on user-defined data types are in the form of procedures and function calls (pp. 22-23).

See the comments on requirement F1.

F5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

....P

Construction	P-
Selection	T
Predicates	P
Type conversions	T
Operations and data can be defined together	T

There is no specific syntax to denote construction of a value of a user-defined type, but the module definition could include a function definition which could return such a value. Selection is accomplished through dot qualification: If T is a user-defined data type, C is a component of that type which has been "exported" (made accessible outside the type definition), and V is a variable of type T, then V.C refers to the instance of the component C in V. The equality predicate is pre-defined for any user-defined type, but it must be explicitly exported before it can be used; other predicates must be defined as functions or procedures. Type conversions can also be defined as functions within the type definition. (See pp. 22-23.)

Adding an explicit construction syntax to the language would cost little. Using the type identifier followed by a list of initial values enclosed in parentheses -- the syntax of a function call but with the advantage that the type name is obvious -- is one possibility (borrowed from CS-4). The appearance of such a construct would cause the execution of a particular routine defined in the type body (again, borrowed from CS-4). Such an implementation would have only a moderate impact on the syntax analyzer of a compiler and almost none elsewhere.

F6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian

products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

....T

EnumerationT
Cartesian products (records)T
Discriminated unionT
Powerset of an enumeration typeT

This requirement is fully satisfied (pp. 17-22, 24-25).

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.D

EUCLID appears to satisfy this requirement. It has no overlay statement and even requires the compiler to check that machine-dependent records (in which the user specifies the bit location of all fields) do not have overlapping fields (p. 24). There are also numerous references to a "non-overlap" property. Unfortunately, there are also the concepts of main variables, entire variables, and part of a variable which we have not been able to fathom, in spite of the report's claim that they are defined precisely (pp. 33-34, 36-37). The language report, in the area where these concepts arise, raises some questions of overlapping which are not resolved.

The scope of any needed changes is impossible to determine, since it is impossible to determine what the needed changes are. Probably only a clear explanation in the report is required.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.T

InitializationT
FinalizationT
Allocation actionsT
Deallocation actionsT

1

EUCLID
Requirement E8

22

This requirement is fully satisfied (pp. 22-23).

F1. The language will allow the user to distinguish between scope of allocation and scope of access.T

EUCLID has the typical Aluol-like nesting of scopes, with the declaration of an identifier in an inner scope overriding any definition in an outer scope. Thus a variable declared (existing) in an outer scope could be inaccessible in some inner scope. It is also possible to declare an identifier to be pervasive in some scope, which forbids another definition of that identifier in any inner scope. Thus the user can specify that the scopes of allocation and access are identical (and not necessarily the entire program). (See pp. 35-36.)

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.P

Allowable operations can be limitedT
Access can be limited where usedPT
External declarations need not all have the same scopeN/A
Naming conflicts can be avoided (renaming)T

These properties of any user-defined data type which are to be accessible in the program must be explicitly declared (exported). Such properties are then fully accessible throughout the access scope of any datum of that type. Access on the user side can be limited through the device of a module; one of the properties of a module is that it forms a closed scope -- any interface with entities outside the module must be explicitly declared (they must be imported). This device satisfies the letter of the Tinman requirement, but it is probably more complicated than the authors of the Tinman envisioned. There are no declarations external to the program; the EUCLID language report does not address libraries or commands. The binding operator, ==, gives a renaming capability, among other things.

The changes necessary to add libraries to the language are discussed under requirements F4 and F5.

F3. The scope of identifiers will be wholly determined at compile time.T

This requirement is fully met (pp. 35-36).

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.F

The EUCLID language report does not address the question of libraries.

Adding a library accessing capability to the language is a minor task. Provided that the capability is not too elaborate, its impact is usually confined to the front end of the compiler.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.F

Program component libraries accessible at compile timeF
Libraries can contain foreign language routinesF
Interface requirements checkable at compile timeF

The EUCLID language report does not address the question of libraries.

See the comments on requirement F4. Requiring that interfaces be checked at compile time would make impact of a library facility somewhat more expensive than those comments suggest, but it could still be done for a moderate cost at most. The greatest drawback to that requirement is that it would require the relevant libraries to be accessible at compile time and would generally reduce the compiler's throughput substantially.

F6. Libraries and Com pools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any

level of programming activity from systems through projects to individual programs. There will be many specialized compilers or libraries any user specified subset of which is immediately accessible from a given program.F

Libraries and compilers will be indistinguishableF
Immediately accessible sublibraries at any levelF

The EUCLID language report does not address the question of libraries.

See the comments on requirement F4.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.F

EUCLID has no such facility.

Adding such a facility to EUCLID, or to any language for that matter, should not be too difficult a task. The hard (and expensive) part is deciding which machine dependent capabilities are common enough and useful enough to warrant inclusion in the language. Once they have been determined, it should be fairly easy to invent a syntax for accessing these capabilities -- particularly so in a language as clean as EUCLID. The costs to the compiler are then a direct function of the elaborateness of the capability -- the number of new syntactic forms which must be analyzed, the number of new code sequences which must be generated, etc.

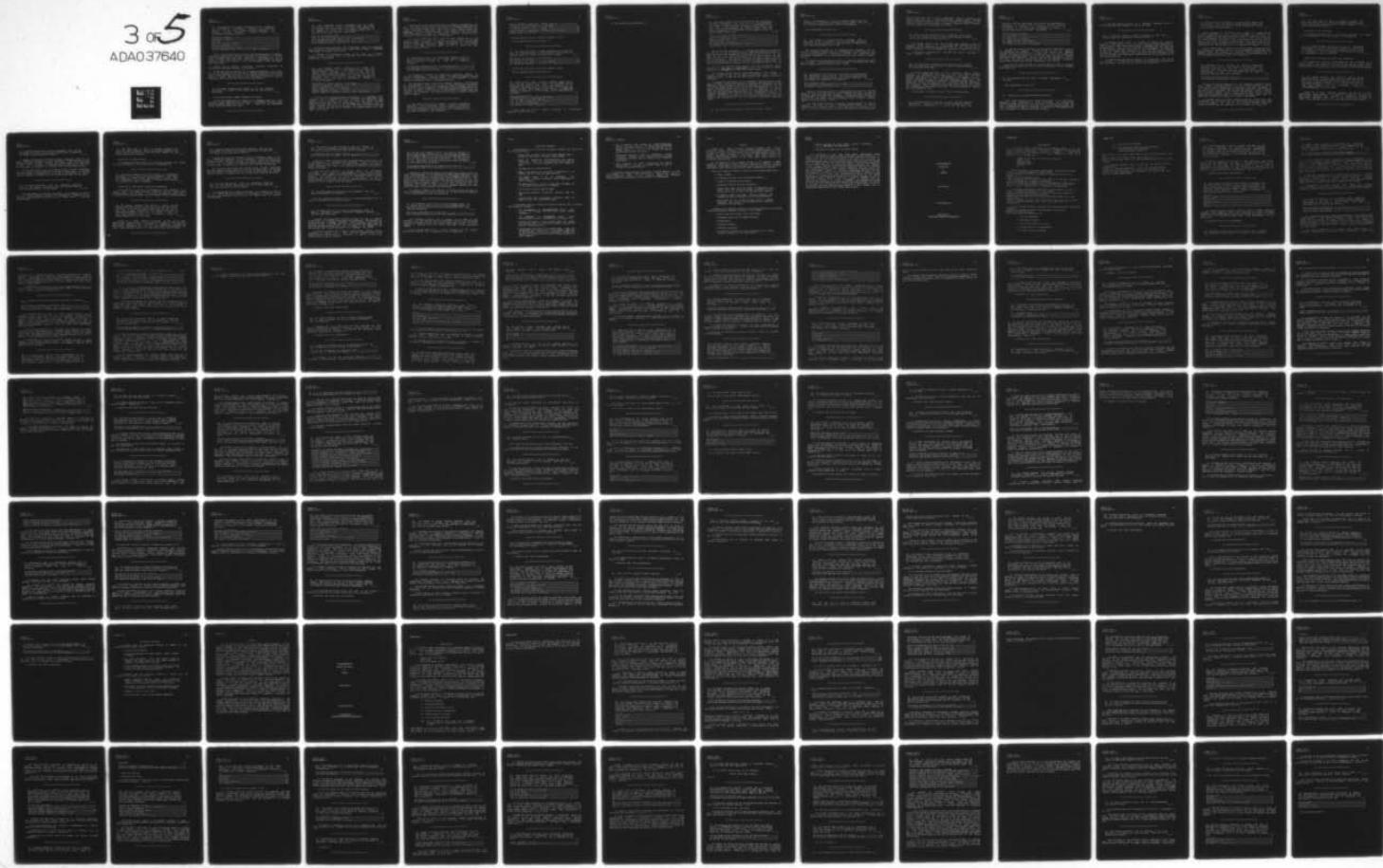
AD-A037 640 COMPUTER SCIENCES CORP FALLS CHURCH VA
DOD PROGRAM FOR SOFTWARE COMMONALITY HIGH ORDER LANGUAGE WORKIN--ETC(U)
1977

F/G 9/2
N00039-75-C-0289

NL

UNCLASSIFIED

3 of 5
ADAO37640



S1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

.....T

Sequential execution	T
Conditional execution	I
Iteration	T
Recursion	T
(Fpseudo) parallel processing	F
Exception handling	F
Asynchronous interrupt handling	F
Control structures from a small set of simple primitives	T

EUCLID has the usual intra-program control structures of sequential execution, conditional execution, iteration, and recursion (pp. 45-53). These structures are formed from a small set of primitives, but can be used to build quite elaborate mechanisms, particularly the looping structures with their multiple exits and parameter value generators (pp. 44, 48-50).

EUCLID has no parallel processing, exception handling, or asynchronous interrupt handling capabilities.

Adding the missing features to the language would be a major task. We do not believe that an good set of parallel processing capabilities has yet been designed, for example. The design effort, then, is a major task in itself. Assuming that whatever the final design is it will be elaborate, the impact on compilers will be substantial.

S2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

....F

EUCLID has neither a goto statement nor labels.

Adding this capability, as stated, to the language would be a minor problem, as the issues are well understood. Even adding a slightly more elaborate goto, permitting transfer from a scope to any containing scope, would cost about the same.

63. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.P+

Based on Boolean expressionT
Based on type from discriminated unionT
Based on computed choice among labeled alternativesT
All alternative must be accounted forF
Simple mechanisms will be supplied for common casesT

EUCLID completely satisfies this requirement, with the exception that the use of a complementary clause (*else after if then and otherwise after case*) is optional (pp. 46-47).

To require the complementary clause in all cases is a trivial change to the language and any compiler, but we consider it a questionable requirement.

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).P+

Termination can occur anywhere in the loopT
Multiple terminating predicates are possibleT
Entry permitted only at the loop headT
Simple cases are clear and efficientP+
Control variable is local to the loopT
Control value is efficiently available after terminationF

EUCLID has two iterative control structures: An "infinite loop" structure and a parameter-controlled structure (a syntactically clean version of the Pascal *for*). For both of these any number of *exit* statements can be used anywhere within the loop to exit to the statement immediately after the loop. In the *for* loop, the termination test is always executed at the top of the loop. The loop parameter in the *for* loop is local to the loop and is treated as a constant within the loop body. (pp. 48-50)

Simple cases of loop termination will be handled efficiently, but they may not be clear because they require a programmed exit statement at the appropriate point in the loop; there are no special syntaxes for common terminating conditions other than those handled by the for statement. The loop control variable is not efficiently available after loop termination; it must be explicitly assigned to some variable accessible in the containing scope before exit.

The cost of adding the missing parts of this requirement to the language would be modest. For example, a simple addition to the loop statement would implement the common head-of-loop and tail-of-loop Boolean terminating conditions (while and until). An equally simple addition to the exit statement could be used to assign the loop parameter only upon exit.

65. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

....P

No recursive procedures within recursive proceduresF
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)F

All routines in EUCLID are potentially recursive, without any explicit declaration. Because of this, and because a routine body can contain the definition of another routine, it is possible to define recursive procedures within a recursive procedure.

The language should be modified to require explicit specification of recursiveness. Once this is done, it would be possible to enforce the constraint against recursive procedures within recursive procedures, if this is truly desirable. In any case, run-time efficiency can be obtained in general if the compiler is not required to assume that all routines are recursive. The actual cost of such a change on compilers would be modest.

66. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

....F

Able to create and terminate parallel processes	F
Process can gain exclusive use of resources	F
No parallel routines within recursive routines	N/A
No routines within parallel routines	N/A
Maximum number of simultaneous instances are declarable	N/A
(Access rules are enforced)	N/A

EUCLID does not support parallel processing at all.

See the comments under requirement 61.

67. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.F

Program can get control for any exception	F
Parameters can be passed	N/A
Can get out of any level of a nest of control	N/A
Can handle the exception at any level of control	N/A

EUCLID does not support exception handling at all.

See the comments under requirement 61.

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.F

Priority specification	F
Synchronization via wait/enable operations	F
Wait for end of real time interval	F
Wait for end of simulated time interval	F
Wait for hardware interrupt	F
(Can enable and disable interrupts)	F

EUCLID does not support parallel processing or asynchronous interrupt handling at all.

EUCLID
Requirement C8

59

See the comments on requirement C1.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....P+

Free format with statement terminator	T
Mnemonic identifiers possible	T
Based on conventional forms	T
Simple grammar	T
No special case notations	P+
No abbreviations of identifiers or keywords	T
Unambiguous grammar	T

EUCLID uses the semi-colon as a statement terminator in general and the end of a line in certain cases. Basically, the end-of-line is a statement terminator whenever the combination of the last token on the line and the next token (the report says the first token on the next line) makes it clear that the last token on the line is indeed the end of the statement. The report then gives two lists of last tokens and next tokens which specify when the implicit end-of-statement occurs. We have not verified that the concept and the specification agree. (p. 13)

The only known special case notations are the conditional escape statements: exit when and return when (p. 44). These could be written as part of an if-then with a negligible extra effort.

The language report contains two descriptions of the grammar: A formally ambiguous one to be read by humans and, in an appendix, an unambiguous one. We have not verified that the two describe the same language.

To remove the conditional escape statements from the language would make compilers simpler and cheaper to implement and maintain, but only slightly. The same could be said about requiring a semi-colon at the end of each statement, but the human factors considerations of making it required only when needed argue against this -- provided, of course, that the semi-colon is indeed required only when necessary and does not introduce any worse human factors problems. Unfortunately, even if the existing definition of when the semi-colon is required is "right", if EUCLID is expanded to anything close to the Tinman requirements it will probably be impossible to keep this feature in any easily understood form.

H2. The user will not be able to modify the source language

syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.T

This requirement is fully met.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.F

The language character set is implementation dependent and the language designers suggest several alternatives, one of which includes the lower-case characters. Lower-case letters, when available, are considered identical to the corresponding upper-case letters, however. (p. 11)

It would be a simple matter to fix on a single character set which is contained in the ASCII 64 character subset. The language designers have suggested ways for doing this (p. 58-59).

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.P

Break character existsP
(Literals are self-identifying as to type)T
(Bit-string literals for any type)P

EUCLID has a break character for identifiers (but does not specify what it is), but not for use within literals (p. 11). Literals of different types have different syntaxes. Numeric (integer) literals may be expressed in octal and hexadecimal notation as well as decimal. There is no prohibition against continuing long character literals beyond the end of a line. (p. 14)

The blank could apparently be used as a break character in numeric literals. The cost of such a feature would be minor. It would be equally simple to require that long character literals be broken at the end of a line. This could be done by specifying that two consecutive

character literals, with possibly intervening blanks, comments, and ends-of-line, are to be concatenated into a single literal. (EUCLID already specifies a special mechanism for denoting the apostrophe -- the character string literal bracket -- inside a character string different from the common "two apostrophes mean one".)

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.T

The language report does not specify that lexemes cannot be continued across the end of a line (other than the character string constant, see the comments on requirement H4), but the implication is strong throughout. This is probably an oversight in writing the report.

A mechanism is specified for including any object character in literal strings (p. 14).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.U

The EUCLID report specifies 61 "word symbols", special symbols comprised of alphabetic characters (p. 12). They are rigorously reserved; they may not be used as identifiers in any context (p. 58). However, throughout the report are references to "standard" entities, which usually have at least one identifier associated with them. The intent of these "standard" entities is unclear (see the comments on requirement F5) and, in particular, it is not clear that the identifiers are reserved. These "standard" identifiers are few enough that EUCLID has relatively few key words in any case. The only question is whether they are reserved or not.

If any change is needed, it would have a negligible cost.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two)

language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

....Pt

Uniform comment conventionT
Look different from codeT
Bracketed by one or two charactersT
Can contain any charactersPt
Can appear anywhere reasonableT
Terminated by the end of the lineF
Compatible with automatic reformattingT

The language report specifies a single comment convention, which is to include the comments between braces ({ and }). However, implementations are permitted to realize the braces with alternate symbols, the ones specifically suggested being /* and */. Comments can contain any character other than the closing brace, can appear anywhere reasonable (not embedded in a lexeme), and are not terminated by the end of a line. They can, of course, look like code if cleverly positioned, but no more so than most comment conventions. (pp. 12, 59)

It would be a simple change to allow the end of a line to terminate a comment. This would also tend to reduce the ability to make comments look like code.

H8. The language will not permit unmatched parentheses of any kind.

....T

This requirement is fully met.

H9. There will be a uniform referent notation.

....Pt

Given that functions cannot appear on the left of an assignment statement, this requirement is fully satisfied. The notation for function references and arrays is identical. In particular, arrays can have structured components, functions can return structured values, and the same notation is used to reference a component of a structured array and a component of a structured function result.

Adding the ability to assign to a function reference would be fairly expensive, and probably not worth the cost.

R10. No language defined symbols appearing in the same context will have essentially different meanings.P

A relatively large number of symbols in EUCLID have multiple meanings. For example, +, -, and * are used as powerset operators as well as numeric ones (representing union, difference, and intersection, respectively). This has some historical justification, but gives rise to the problems expressed in the text of this requirement. In addition, <= and >= denote set inclusion as well as the obvious numeric relationals, and the latter is even used for logical implication! (This last is a fluke of an "incorrect" specification of the representation of the Boolean values.)

To remove these multiple use symbols is an easy task. They should be retained for their numeric meanings and new key words invented for the other uses. The cost of such a change is negligible.

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

....P

EUCLID contains a relatively large number of implementation dependent defaults for such a modern language. One section of the language report (section 13) specifies minimums for some of these, but the implementation dependence is still present. Few of these defaults will affect program logic (the ranges of the signed and unsigned integer types, which will be fixed by the hardware of the object machine, are exceptions), but the dependence does mean that programs which compile correctly on one compiler may have to be modified to compile on another.

It is a simple task to specify the default values, and to eliminate the signed and unsigned integer types (requiring an explicit range specification). The problems of enforcing these specifications by assuring that no compiler exceeds them might be insurmountable, even in the relatively tightly controlled DOD environment.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

....P+

Defaults specified for don't care casesP+
Programmer can override the defaultsP+

EUCLID does provide defaults for the generally accepted "don't care" cases: Data representation, open or closed subroutine expansion (but not reentrant or nonreentrant code), etc. With most languages it is possible to find places where an override capability might be wanted and EUCLID is no exception. For example, the language report suggests that compilers should insert "small" routine bodies in line, but the programmer who disagrees with the compiler's judgment about what is small has no ability to require the routine to be closed.

The hardest part of modifying EUCLID to make it comply with this requirement is identifying all the "don't care" cases which might need to be overridden. The cost of the necessary changes cannot be determined until they are identified, but it should not be too great.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

....F

EUCLID has no such facility.

The necessary addition would be fairly easy, provided the feature is not too elaborate. See the comments on requirement 14.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

....F

EUCLID has no conditional compilation capability.

This facility can usually be implemented in most compilers at a modest cost, the cost depending primarily on the source level elaborateness of the facility. If there are any compilers which need to be changed, the changes usually can be isolated in their initial phases.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

....I

EUCLID is a small, powerful, language, as far as it goes. Furthermore, there seems to be a concept of a kernel of the language, with various entities being described as "standard", although the identifiers associated with them do not appear in any reserved word list. These entities appear to be extensions to the base language.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

....P

A number of translator dependent limits -- including some of the ones specifically mentioned in this requirement -- are not fixed by the language report. However, minimums for these limits are specified (p. 58). It is possible to complain that at least one of these (the maximum number of elements in a powerset --16) has been set too low.

It is an easy task to fix these limits in the language definition and the effect on most compilers would not be great, but -- as for the compiler defaults (Requirement 11) -- it may not be possible to control compiler writers to prevent them from treating the limits as minimums.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....P+

This requirement is basically satisfied. It is possible to look at the low "limit" (minimum maximum) of 16 elements in a powerset (p. 58) as having been dictated by the word size of a large number of existing minicomputers.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

....F

EUCLID has no such facility.

The necessary addition would be fairly easy, provided the feature is not too elaborate. See the comments on requirement 14.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

....F

EUCLID has no conditional compilation capability.

This facility can usually be implemented in most compilers at a modest cost, the cost depending primarily on the source level elaborateness of the facility. If there are any compilers which need to be changed, the changes usually can be isolated in their initial phases.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

....I

EUCLID is a small, powerful, language, as far as it goes. Furthermore, there seems to be a concept of a kernel of the language, with various entities being described as "standard", although the identifiers associated with them do not appear in any reserved word list. These entities appear to be extensions to the base language.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

....P

A number of translator dependent limits -- including some of the ones specifically mentioned in this requirement -- are not fixed by the language report. However, minimums for these limits are specified (p. 59). It is possible to complain that at least one of these (the maximum number of elements in a powerset --16) has been set too low.

It is an easy task to fix these limits in the language definition and the effect on most compilers would not be great, but -- as for the compiler defaults (requirement 11) -- it may not be possible to control compiler writers to prevent them from treating the limits as minimums.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....Pt

This requirement is basically satisfied. It is possible to look at the low "limit" (minimum maximum) of 16 elements in a powerset (p. 58) as having been dictated by the word size of a large number of existing minicomputers.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.U

No efficiency cost for unused featuresU
Efficient code can be produced for all featuresF

EUCLID is a clean language and the designers have given quite a bit of consideration to implementation questions. There is reason to believe that a compiler can be built which would not charge the user for unused features, but there is no compiler in existence at present to verify this belief.

There are some features of EUCLID which are potentially inefficient, and these are acknowledged in the language report (pp. 37-38). Such potential inefficiencies are in almost any high-level language, and getting rid of them is impossible within current technology, at least if the language is to be useful. It is the responsibility of the user to learn the inefficiencies of the language and decide if the feature should be avoided.

J2. Any optimizations performed by the translator will not change the effect of the program.U

This is a compiler requirement, not a language requirement, and is thus not addressed by the language report.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.P+

A machine code capability is defined through the use of machine instructions in routines (procedures and functions). The routines can be declared with an inline attribute, which requires that they be expanded in line. Thus the user has a full machine code capability, but with some clumsiness involved if all he wants is to insert a few instructions into a single location in his program. (pp. 51-57)

It would be a simple job to invent a syntax which permits in-line insertions of machine code. There would be almost no cost to any compiler because the ability to recognize and compile machine code instructions is already present.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

....P+

Encapsulated specification of representation possibleT
Space/time tradeoff can be specifiedP

EUCLID has a machine-dependent record capability, which permits the exact specification of the positions of fields within records (p. 24). Other data structures may be specified with the packed attribute, which directs the compiler to conserve space at the expense of time (p. 19). There is no attribute which directs the compilers to generate the most time-efficient structure, however. Thus if the compiler's default data structure requires overly clumsy code for some particular use, the programmer is forced to rely on machine-dependent records, and these sometimes result in inefficient code themselves.

It is easy to invent a new attribute to further refine the types of data layouts generated by the compiler, but the expense can be moderate to high, particularly in the code generator.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

....P

Open/closed properties can be specifiedP
Open and closed versions have the same semanticsT

It is possible to specify that a routine is to have an open expansion and the language report specifically calls for no difference in semantics between the two types of expansion (pp. 51, 54). It is not possible to specify that the routine must have a closed expansion, however.

It is a trivial task to add a closed attribute to the routine declarations and the cost to compilers would be negligible.

Extraneous Features

We recommend that the following features of EUCLID, not required by the Tinman, be kept:

- * String type (assuming that the Tinman requires only a character type, not a character string type).
- * Octal and hexadecimal representations for numeric constants. These are occasionally useful and cost very little. A binary capability might be added as well.
- * The mod operator (remaindering).
- * Zone, which allows the programmer to implement his own dynamic storage allocation when desired.
- * The elseif clause of the if statement. This implements a common structure often overlooked in other languages.
- * The exit statement. This is a surrogate for goto, but it has useful documentation attributes.
- * The numeric functions abs and odd.
- * The numeric-character conversion functions chr and ord.
- * The successor and predecessor functions, succ and pred, for use with enumeration types.

We recommend that the following features of EUCLID, not required by the Tinman, be deleted:

- * The storageUnit and storageUnitBits types. Their purpose is unclear and they are highly machine dependent.
- * The signedInt and unsignedInt types. Range specification should be required of all integer data.
- * Legality assertions. These conflict with the Tinman requirement that all features be within the state of the art (requirement #1).
- * Specification of internal representations. These are incomplete and even "wrong". For example, True must be represented by binary zero and False by binary one, exactly the reverse of the "natural" representation on many computers.

- * The requirement that fields in machine-dependent records must not overlap. To remove this requirement permits all sorts of mischief, but the ability to do strange things is one of the reasons for needing user-defined record structures.
- * Restricting pointers to point to dynamically created variables. Removing this restriction gives up some protection, but that loss is offset by the increased capability.
- * Dual notations for field referencing and pointer dereferencing. Multiple notations lead to lack of clarity in program listings.

In addition, EUCLID defines an unchecked type-conversion (p. 31). This feature is useful, even necessary in some cases, but it is machine-dependent. Uses of this feature should be bracketed (encapsulated) by statements which point out that dependence.

Summary

EUCLID is a small, elegant, language designed for writing verifiable system programs. The designers have been rigorously guided by those dual goals -- verifiability and system programming -- and have produced a spare language, yet one which is quite powerful within its tightly circumscribed limits. They have also been guided by implementation considerations, with the result that EUCLID appears to be eminently implementable.

We say "appears to be" because EUCLID is at present a paper language -- there are no known compilers, although at least one is being developed. Thus there is no experience to back up any of the opinions expressed in this report, particularly those concerning efficiency.

Among the "unusual" features required by the Tinman, EUCLID has:

- * Strong typing.
- * A complete, regular, data definition facility.
- * A lack of arbitrary restrictions.
- * Recursive routines and data structures.
- * Scope rules which permit the scope of allocation to be larger than the scope of access, but which can also force them to be the same, yet not the entire program.
- * A useable pointer mechanism, although probably more restrictive than is desirable in the DOD environment.
- * Free format of the source program with a context sensitive rule which permits omitting the statement terminator in many cases.

Because of its size and conflicts with the two design goals, EUCLID lacks the following features, required by the Tinman:

- * Fixed point and floating point data types.
- * Statement labels and the goto statement.
- * Input/output.
- * Exception handling.
- * Parallel processing.
- * An ability to define new infix operators or to extend existing operators to new data types.

- * Special syntaxes for some common control structures (e.g., loop while and loop until).
- * Procedure and exception parameter classes.

It is difficult to keep from being overly enthusiastic about EUCLID. The designers have produced a powerful, elegant, language. Of course, it is to be expected that problems will arise as the language is implemented, but we do not believe that they will be major. As mentioned above, many of the missing features have been left out because they conflict with the two basic design goals. I/O and fixed point and floating point types are not needed in system programming, for example. A large number of these missing features could be added to the language with little difficulty, as they present compiling problems which are well understood. Others -- parallel processing and infix operator definition, for example -- are not so well understood (at least as far as agreeing on a "best" set of features or syntax) and could cause problems. Any attempt to enlarge EUCLID to bring it up to the Tinman requirements will doubtlessly result in a clumsier language, and one at odds with EUCLID's goals. (For this reason we suggest that the verification features of EUCLID be dropped. A Tinman-sized language will have features which make verification impossible within the state of the art.) Yet it appears that, given such a solid basis, if such an enlargement were done with great care there is a good chance that an appropriate realization of the Tinman would result.

A COMPARISON OF
JOVIAL J3B
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language JOVIAL J3B to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, JOVIAL J3B is considered to be defined by:

JOVIAL/JZP Language Specification Extension 2
Softech, Inc.
Waltham, Mass.
Document 2744-4.2
July 1975

Tinman contains 79 language requirements. This report compares JOVIAL J3B to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used.

(Occasionally no text is needed if a requirement is totally satisfied.)

If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P-, P+, and F- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of JOVIAL J3B which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring it into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....T

JOVIAL/J3R requires that the type of each variable, data table, array, etc. be explicitly specified by the user at the time of data definition. All items must be declared before use. The type of operations and expressions are not specified by the user, but are determinable at compile time. The type of any item cannot be changed at run-time. Hence J3R meets this requirement. (pp. 4-3, 4-14)

No conflict with other requirements.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....T

Integer	T
Floating Point	T
Fixed Point	T
Boolean	T
Character String	T
Arrays	T
Records	T

J3R allows signed or unsigned integers, single or double precision floating point numbers, fixed point numbers, bit strings (Boolean variables), character strings, arrays, and tables (records), as built-in data types. It therefore fully meets this requirement. (p. 1-2, Section 8).

No conflict with other requirements.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic

JOVIAL J3B
Requirement A3

2

and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.P-

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedP

J3B does not require that the precision for all computations to be performed in floating point arithmetic be declared globally for any scope. Only individual specification of single or double precision floating point variables is permitted. The actual precision is dependent upon the word size of the computer and its representation. The precision specification for fixed point numbers is also machine dependent and is not controlled by the user. (pp. 1-5, 1-13, 1-14, 2-2, 2-10, 2-11, 6-2)

To meet this requirement the J3B language will have to be modified to require that precision for all floating point arithmetic be declared globally. Furthermore, it should permit explicit precision specification for individual floating and fixed point variables rather than making them hardware word-size dependent as at present.

Implementation of these changes will affect all existing implementations. Expression processing, data definition, allocation procedures, and the syntax checking mechanism will have to be modified.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.P

Treated as exact quantitiesP
Range and step size determined at compile timeF
Scaling handled automaticallyT

"most representations for fixed point numbers do not permit exact representation of many decimal fractions. Hence, a JOVIAL/J3B literal value may not denote its true decimal value but rather an implementation-dependent approximation to its true value in the set of <FIXED(A)> values. The range and step size for fixed value variables are not specified by the user either. Scaling is handled automatically. (e. 1-14, 2-11.)

The exact representation of a decimal number into binary is a practical impossibility for existing types of computers. However, more

JOVIAL J3E
Requirement A4

visibility can be allowed the user by providing truncation and rounding routines (e.g., round up, round down, round even, etc.) in the language library. Furthermore, the language should permit user specification of the level of precision for these numbers. Range and step size should also be explicitly specified in the language syntax for fixed point variable declarations.

Implementation of these changes in the language features will have some impact on all existing compilers in the areas of syntax analysis, library routines, storage allocation, etc.

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)N/A

J3E does not allow the user to specify and enumerate his own character set. The user must use the standard language-defined character set consisting of 26 upper case alphabetic characters, the decimal digits, and 1% special characters. The provision for ASCII or EBCDIC code for the character set and conversion between two sets of codes is not controlled by the user and is completely hardware and installation dependent. Hence, J3E does not meet this requirement. (pp. 2-2, 2-3.)

To meet this requirement the language must provide for status variables and allow enumeration of character sets defined by the user. The library must support conversion routines between ASCII and EBCDIC which must be available to the user. The user program should then be allowed to use the user defined character set.

Implementation of these changes in the language will have a major impact on all existing implementations. The entire lexical and syntax analysis phases of the compiler (string manipulation, comments etc.) will have to be modified.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will

be determinable at entry to the array allocation scope.P

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryF
Subscripts only integers or from an enumeration typeT
Subscripts will be from a contiguous rangeT

The J3P language specifications require that the user specify one, two or three dimensions for an array at compile time. The type of the array is fixed at compile time. All elements of the array are required to be homogeneous in type. The language automatically fixes the lower bound of the array to 0. The user must specify the upper bound of the array at compile time as a positive integer constant. The upper bound is not evaluable at scope entry time. (Section 4.4.)

To meet this requirement the language will have to allow the use of expressions in upper subscripts which evaluate to a positive integer at scope entry time. The arrays will then have to be allocated at execution time instead of at compile time as is presently done in the existing implementations.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.P

Alternative structures for records are possibleT
Discrimination condition may be any Boolean expressionF

JOVIAL/J3P provides an OVERLAY feature which allows the user to specify not only the records or variables which can overlap each other, but also the names of variables, arrays and records which are assigned successive storage locations in core. For example, OVERLAY A,B,C=F1 will allow allocation of A, B, and C to be contiguous, one after another, while at the same time F1 will be allocated starting at the location where C was allocated. Each variable will be available to the user at run time. However, the language does not provide for a Boolean expression to be used as the discrimination condition to be used for field selection. Hence it does not meet that portion of the requirement. (Section 4.6.)

To meet this requirement the language syntax will have to be modified to allow selection of overlaid fields after evaluating a Boolean expression. The result of the evaluation will determine which of the overlaid fields is to be selected and returned to the user.

JOVIAL JPF
Requirement A7

5

The existing compilers do not provide such facility and will have to be modified to accommodate this change in the language.

JOVIAL JZP
Requirement B1

6

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.P

Automatically defined for any type (except...)P
Available for individual componentsT
(Assignment and reference via functions)T

Assignment and reference operations are defined for the built-in data types in JZP. Values for a built-in type can be assigned to the variable, pseudo variable, array component or record component of that type. Additionally, conversions are allowed to assign values of a different type to a target of built-in data type. The assigned values can be referenced by using the name of the target. However, the assignment operation is not defined for conformable arrays and records as a whole. (Section 5.2.)

To conflict with other requirements.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.T

JZP provides for = sign which can be used for comparing two data objects (regardless of type) for identity. Logical and not bit by bit comparisons are made for equality. Conversions are performed before comparison as necessary. (Section 7.)

To conflict with other requirements.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P

Built-in for all numeric and enumeration typesP
Ordering can be inhibited when desiredP

JZP provides for all six relational operators (n. 7-12) for comparison of numeric data. However, since the language does not allow

JOVIAL J3P
Requirement R3

7

user-defined data or status variables, these operators are not defined for enumerated data or user-defined data. For the same reason, no mechanism is available in the language to inhibit ordering of data.

To meet this requirement the language definition will have to be extended to provide for user definition of new data types and status variables. Then language mechanisms should be provided to enumerate data in either an ordered or unordered way.

A major modification to existing implementations will be necessary to accommodate these changes in the language definition. Type checking mechanisms, syntax analyzer, dictionary and code generation phases will be affected.

R4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.Ft

Addition	T
Subtraction	F
Multiplication	T
Division with real result	T
Exponentiation	T
Integer and fixed point division with remainder	F
Negation	T

of these arithmetic operations, J3P provides for all of them except integer division with remainder. (Sections 2.2.2, 7.1.)

The language syntax will have to be modified to provide for integer division with a remainder. The user should be able to specify a location where the remainder can be stored.

This change will have a minimal impact upon existing compilers in the areas of expression processing and syntax checking.

R5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the

specified precision will be allowed for floating point numbers.F

Never from the left for data within rangeF

Never on the right for integer and fixed pointF

Implicit floating point rounding beyond precision allowedF

(Run time checks can be avoided)N/A

J3F does not specify rules for truncation and leaves that implementation dependent (p. 7-7). Therefore there is no guarantee that truncation will take place only on the right. Moreover, the specifications state that for fixed point computation, implicit truncation of precision bits on the right may occur (c. 7-10). Even the floating point rounding for both single precision and double precision numbers is implementation dependent. Hence the language completely fails to meet this requirement.

The language specifications should be modified to require all truncation to take place on the right and with warning to the user. The precision specification for fixed and floating point number should be allowed to the user. The language should permit implicit rounding beyond the precision specified by the user.

These changes in the language will require minimal changes in the existing compilers. Most existing implementations attempt truncations from the left. The areas affected will be data declarations, expression processing, and run-time diagnostics.

86. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P+

short-circuit andP

short-circuit orP

NotT

XorT

J3F provides for AND, OR, NOT and XOR Boolean operators. It permits but does not require, the short circuit evaluation of these Boolean operators. (p. 7-14).

To meet the requirement a minimal change in the language definition will have to be made to require short circuit evaluation of Boolean expressions. It will have minimal effect on those implementations of the language which do not evaluate Boolean expressions in short circuit mode.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.F

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF

The language definition does not specify that scalar operations and assignment are permissible on conformable arrays, nor does it allow data transfers between arrays and record of identical logical structure. Any such data transfers or scalar operations between conformable array and record structures are, therefore, implementation dependent. (Sections 4.3, 5.2.)

The scalar operations for addition, subtraction and logical comparison can be added to the language definition for conformable arrays. The assignment operation can be extended to allow data transfer between conformable records and arrays. But the extension of multiplication and division operations to conformable arrays may be meaningless.

Some of the existing implementations may have to be modified to support this extended definition of scalar operators for arrays and records.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.F

No implicit conversionsF
Explicit between integer, fixed point, and floating pointF
Explicit between fixed point scale factorsF
(Explicit between integer and Boolean)F
(Explicit between integer and enumerated types)F
(Explicit between different enumerated types)F

J3P allows implicit conversions between integer, fixed point and floating point numbers and permits mixed mode arithmetic.

To provide for explicit conversions from one data type to another the language definition will have to be modified. Built-in functions or other language features will have to be provided to perform explicit conversions from one type of data to another. Similar functions will also be required to handle fixed point scale factors.

Implementation of these capabilities will affect all existing implementations. Major changes will have to be made in the areas of expression processing assignment and syntax checking.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

....P

Implicit conversion between rangesF
Exception condition on integer and fixed point truncationI

J3P does not allow range specification for variables. It does, however, require an error condition to be raised if the value of an integer or fixed constant exceeds the value that can be stored in the computer words assigned to store that value (usually an integral number of full computer words is assigned). (Sections 7.2.3.1, 8.1.1, 8.1.4)

The language modification to provide for user specification of range of variables is necessary. Explicit conversion operations between ranges will not be required.

This language modification will have minimal impact on existing implementations.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

....F

Sending and receiving of dataF

Sending and receiving of control information	F
Dynamic device assignment	F
User exception condition control	F
Installation independence	F
(Data formatting capability)	F
(Reading and writing of bit strings)	F

J3R does not provide for I/O features. Hence there are no language constructs which allow program interaction with files, channels, devices or terminals. No capability to read or write data on I/O devices is available and no static or dynamic assignment of I/O devices is permissible. The user can not control exception conditions either. All of these features are implementation-dependent.

A systematic, standardized set of I/O statements needs to be defined for the language to allow it to perform each of the functions listed in the requirement. It calls for introduction of such constructs as READ, WRITE (with or without format) OPEN, CLOSE, ON OVERFLOW etc. to accomplish this.

The existing implementations have defined their own non-standardized I/O features to perform some of these functions. Implementation of well-defined standardized features may require significant modifications to existing compilers.

R11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.F

Union	F
Intersection	F
Difference	F
Complement	F
Membership predicate	F
(Set inclusion)	F

The language provides the operations of OR, AND, A-B, NOT and XOR to be performed on bit strings including the built-in data types. However the language does not provide for status variables and hence these operations cannot be applied to enumerated types or their power set. Hence the language only meets this requirement partially. (Section 7.3)

The language will have to provide a mechanism to define status variables and the enumerated types. The definition of these operations

will have to be extended to apply to the power set of these enumerated types.

The existing implementations will have to undergo a moderate amount of change in the areas of data definition, expression processing, IF and FOR-WHILE statement processing, etc. to accommodate the full impact of the language modification.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.T

Side effects must occur in left-to-right orderT
(Embedded assignments)F

JSL provides a detailed set of rules for evaluating expressions. These rules encompass the order of evaluation for different data types and operators and also list rules for evaluation of subscripts, functions, functional modifiers, parentheses etc. In cases of operators having equal precedence, the specifications state that the order of evaluation will be from left-to-right. (Section 7)

No conflict with other requirements.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.T

Few precedence levelsT
No user-defined precedence levelsT
Operands of an operation are obviousT

The language establishes the hierarchy for evaluating expressions for data types to be: (1) double float (2) single float and (3) integer. The order of evaluation for operators is also defined and is as follows: (1) exponentiation (highest), (2) multiplication and division, and (3) unary and binary + and - (lowest). When two operators with equal precedence level are encountered, they are evaluated from left to right. However, the order of evaluation of constituent parts of the expression is implementation dependent. Hence, the language meets this requirement. (Section 7)

No conflict with other requirements.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.T

J3P allows expressions to be used wherever constants and variables of a type are allowed.

No conflict with other features.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.f

In the data declaration portion of the language, substitution of a constant expression in place of a constant value is not allowed in J3P. For instance, in the array declaration the dimensions of the array must be integer constants and not integer constant expressions. The same is true for constants in ITEM declarations, TABLE declarations, etc. (Section 4.3, 4.4, 4.5).

The language definition should be modified to allow constant expressions wherever constants are permitted in the data declarations or other features of the language.

This modification to the language will require minimal changes to existing compilers in the areas of lexical and syntax analysis. Storage allocation for various types of data at compile time will have to wait under after a call to expression processor which will resolve all constant expressions into constants.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.f

Parameter rules consistent in all contextsf
No special operations applicable only to parametersl

J3P allows parameters for procedures, built-in operators and for declarations but does not permit type parameters, exception handling parameters and parallel processing parameters. The language however, does not permit special operations applicable only to parameters.

Language capabilities for type declaration exception handling and parallel processing will have to be added. This would require significant change to existing implementations.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.ft

Actual and formal parameters will agree in typeT
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedF

JSP requires that the type of actual and formal parameters be the same. It also requires the number of dimensions to be fixed at compile time. However, it also specifies that the values of the dimensions (i.e., the subscript range of the arrays) between actual and formal parameters agree. Thus it does not allow the size and subscript range to be passed as a parameter by the user.

If the language definition relaxed the rule requiring agreement between the values of dimensions of the array in the formal and actual parameters at compile time, then these values can be passed as parameters, thus meeting the Tinman requirement.

Modification of this language feature will require minimal changes in type checking and parameter checking mechanisms of the existing compilers.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P

Act as constants (call by value plus)T
Act as variables (call by reference)T
Exception controlF
Procedure parametersF
(Act as variables, but call by value)T

(Act as variables, result parameter)F

JSP allows call by value and call by reference but does not provide for exception control parameters, nor does it provide for procedures to be included in the call as actual parameters.

The J7B language definition should be extended to allow exception handling and also exception control parameters (e.g., labels) in the procedure calls. Procedure should also be allowed to be included in these calls.

These modifications to the language will constitute a major change to existing compilers. Exception handling itself will require changes or extensions to several existing language constructs.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.F

Above properties optionalP-
Above properties are fixed at run timeF

The specification of range and format is not permitted in JSP formal parameters. If a formal parameter is a constant its scale and precision can be specified by the user but there is no guarantee that the precision will be maintained by the translator. Truncation and/or rounding will occur depending upon the word size of computer. The number of dimensions as well as their values are required to be supplied by the user and is not optional.

To meet this requirement several changes in the language definition and syntax shall have to be made. The formal parameters should be allowed to optionally contain definitions of type, range, precision, scale and format. None of these options are presently allowed in an explicit form. Moreover, dynamic arrays should be permitted by allowing the dimension specification to be optional.

Implementation of dynamic arrays will require many changes to existing implementations. Other changes listed in the previous paragraph will significantly affect the syntax checking and code generation phases of the existing compilers.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

....P-

Variable number of arguments possibleF
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeT

J3R does not provide for a variable number of arguments in procedure calls or in procedure definitions. There is no restriction on the number of arguments which must be of the same type. The number of arguments in each call must be fixed at compile time.

To meet this requirement the language will have to provide for moderate changes in language syntax. However, significant changes will have to be made in the parameter passing mechanisms of the translators to implement this change.

D1. The user will have the ability to associate constant values of any type with identifiers.T

The CONSTANT declaration allows a name to be associated with a constant value. (Section 4.7)

No conflict with other language features.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesT
Consistent interpretation in program and dataF

J3R provides detailed description of the properties of each type of literal (Section 1.7.3.1). The syntax for these literals is described in Section 3 in detail. There are, however, no I/O facilities described for the language nor is there a description of the relationship of input data to literals.

I/O facilities and data format description should be provided to meet this requirement.

Implementation of data format which is identical to the format of data literals will require minimal change in existing implementations.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesF

The language permits initialization of items, tables, arrays, characters, bits, pointers etc. at the time of their definition. The actual allocation of these initial values occurs at scope entry time.

The language, however, also permits initialization of variables by default. For instance, uninitialized elements of the array or table are set to the value equivalent to all binary zeroes. (Section 4.9)

The language should forbid initialization of variables by default to meet this requirement. This would lead to two possible alternatives: either issue a warning whenever a variable, table or array is used before initialization or require that all variables, tables, arrays etc., must be explicitly initialized by the user at the time of their definition. Issuing a warning if a variable is used before being assigned a value is difficult, if not impossible in some cases.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.F

Numeric variable range specification mandatoryF
Fixed point variable step size specification mandatoryF
Range and step size specifications do not define a new typeT

Specification of ranges of variables is not mandatory in J3E, nor is there a language option available to explicitly specify the range. The range is implicitly dependent upon the word size of the machine. The same arguments are applicable for fixed point variables. No explicit option is permitted to specify the step size of fixed point variables. Thus the language does not meet this requirement.

Explicit language options should be provided to specify range of variables and arrays. Similar options for step size specification should be provided. Implementation of these facilities will require minimal changes in the syntax checking, code generation and storage allocation functions of the existing translators.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.F

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataF

J3P does not allow explicit association of a range of values with variables, arrays, or components of a record. It does not allow defined type or enumeration of data and hence also does not permit their association with range of values.

The language restricts arrays to be declared as part of data tables (records) (L. 4-+) and vice versa. Hence it fails completely to meet this requirement. (Sections 4.3, 4.4)

To fully meet this requirement, the language will require major modifications to its definition. It must first allow user definition of new data types, enumeration of data via status variables, and permit records in array and arrays in record definitions. Then it should allow specification of a range of values for each of these cases.

All existing implementations will have a major impact as a result of these changes.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.P

Recursive and network structures providedT
Handles variable-value and structure-component connectionsT
Pointer property is an attribute of a typed variableF
Pointer property not for constants, affects only assignmentT
Pointer property mandatory for dynamic allocationF
Allocation scope never wider than access scopeF
(Either the value or the pointer is modifiable)T
(Pointer mechanism handles procedures and parameters)F
(Remap and replace assignment have different syntaxes)F
(Built-in dynamic variable creation)F
(Variable equivalence classes are declarable)F

The language provides for pointer actions which can be used to share data values as well as to build recursive data structures. The pointer mechanism can be used to connect a pointer variable with a value as well as to a structure or to its component. However the language does not ensure safety while using this mechanism. It is up to the user to ensure that no values are accessed which are beyond the scope of the

pointer variable. No dynamic allocation of variables is allowed. Hence the language only partially meets the requirement. (Section 8.4, 7.7, 7.4)

To meet this requirement the pointer mechanism will have to be extended and included as part of the type definition. To provide all the capabilities listed in the requirement will be a major change in existing implementations.

E1. The user of the language will be able to define new data types and operations within programs.F

JZB does not permit definition of user-defined data types or operations.

The language must be changed to allow a mechanism to define new data types and operations. Once defined, these data types and operations should be given the same treatment as the built-in data types. This includes the language's ability to permit initialization, specification of actions to be taken at termination, capability to allocate and deallocate these defined data types. Defined types should be allowed to be formed from built-in types via enumeration, cartesian products, discriminated union and powerset of enumeration type.

Implementation of this definitional facility and the associated type checking required will be a major undertaking affecting syntax analysis, dictionary entries, code generation, diagnostics} etc.

E2. The "use" of defined types will be indistinguishable from built-in types.F

Since there are no defined types, this requirement is not met.

As explained in E1, the language should permit definition of new data types and make them indistinguishable from built-in types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.T

JZB requires that each program component be defined and declared in the program. Items, arrays, tables, procedures, etc., must be declared at SYSTEM, COMPOOL, GLOBAL or LOCAL levels. The compiler will not substitute definitions for undeclared names by default. Instead it will give a compile-time error.

No conflict with other languages features.

E4. The user will be able, within the source language, to extend existing operators to new data types.F

Since the language does not permit defined types it does not meet this requirement.

For the scope of the needed modifications, see F1.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.F

Construction	F
Selection	F
Predicates	F
Type conversions	F
Operations and data can be defined together	F

The J3E language does not provide the facility for user defined data types and declaring the list of operations applicable to that data types.

This is an extension of the facilities discussed in E1. Provision should be made to define the applicable operations for defined data types. These can be built-in operations or user-defined operations.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the powerset of an enumeration type. These definitions will be processed entirely at compile time.F

Enumeration	F
Cartesian products (records)	F
Discriminated union	F
Powerset of an enumeration type	F

J3P does not support defined types at all.

For the scope of the needed modifications, see E1.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.F

The OVERLAY feature of J3P provides free union, among other things.

To remove the OVERLAY capability would greatly simplify existing compilers, but would have a profound effect on existing programs written in J3P.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.F

InitializationF
FinalizationF
Allocation actionsF
Deallocation actionsF

J3P does not support defined types at all.

For the scope of the needed modifications, see E1.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.1

The J3E language clearly defines and distinguishes between four types of allocation scopes: SYSTEM, COMPOD, GLOBAL, and LOCAL. It also specifies the access scope for variables declared within each of these scopes including the rules for accessing the appropriate variable when two variables with the same name are allocated in that scope. (Section 3.2)

No conflict with other languages features.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.2

Allowable operations can be limitedF
Access can be limited where usedP
External declarations need not all have the same scopeF
Name conflicts can be avoided (renaming)F

J3E does not allow the capability to limit access to structures where they are defined. For example, structures defined as GLOBAL are accessible from all components of a program. However, the access to these structures can be limited on where used by redefining the structure at local level. In such cases the global value will be inaccessible.

J3E does not permit a renaming capability to allow two or more names for the same entity.

Additional language capabilities will have to be introduced to specify the scope of a variable at the time of its definition and also at the time of its use. This facility must also be extended to defined data types.

A renaming capability to establish equivalence between several different names should also be added.

Introduction of these changes will effect all existing compilers.

F3. The scope of identifiers will be wholly determined at compile time.T

J3F establishes the scope of each identifier at the time of its definition. See F1 and F3.

No conflict with other language features.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.T

J3F provides for comports and library capabilities. It defines certain built-in functions (called functional modifiers) in the language and permits the user to add as many application oriented functions and procedures as necessary. User can also define data in various comports. Hence the language meets this requirement.

No conflict with other language features.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.Pt

Program component libraries accessible at compile timeT
Libraries can contain foreign language routinesP
Interface requirements checkable at compile timeT

J3F permits library procedures to be accessible to the user at compile time, utilizing either INLINE or COPY options. It also permits interfaces with assembly language routines, but does not specify if interfaces with compiled routines originally written in other high order languages is permissible or not. Hence it only partially meets this portion of the requirement.

J3P requires all interfaces within the library and other procedures to be checked at compile time.

To fully meet this requirement the language definition will have to be extended to specify how J3P will interact with routines written in other high order languages. The simplest way to meet this requirement will be to allow interfaces with compiled versions of these languages, requiring J3P compilers to utilize header information in compiled programs to be used for establishing the interface.

F6. Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.P+

Libraries and compools will be indistinguishableP
Immediately accessible sublibraries at any levelT

J3P permits both compools and libraries. Compools can be used for holding data definitions, while libraries can be used for storing procedures and functions. A project can create its own set of compools and libraries. However, the two concepts of compools and libraries are distinguishable from each other. Even their invocations in programs are different. (A declaration from a compool is accessed via COMPOOL construct (Section 3.1.2) while a library routine is accessed via a call statement (Section 5.3.1) consisting of procedure name followed by actual parameters in parentheses.)

To make the compool and library indistinguishable to the user a simple change can be made in the language syntax to make the two types of accesses identical to each other. Implementation of these changes without altering the actual implementation of compools and libraries can be accomplished with minimal effort. However, to combine the compool and library in the compilers and also make them indistinguishable to the user may require changes in compiler design of moderate proportions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.T

The language provides interfaces with machine dependent capabilities via preprocessed compools, assembly language code, and

certain functional modifiers (e.g., SHIFTR, SHIFTL etc.). All machine dependent capabilities including I/O and interfaces with peripheral and special hardware must be coded in assembly language procedures. These routines can then be interfaced with J3R programs via a call to these procedures. Hence the language meets this requirement.

No conflict with other language features.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.P+

Sequential executionF
Conditional executionF
IterationF
RecursionF
(Pseudo) parallel processingF
Exception handlingF
Asynchronous interrupt handlingF
Control structures from a small set of simple primitivesF

The language provides for sequential execution of statements, has IF-THEN-ELSE and SWITCH conditional control statements and FOR iteration loop. It does not provide for recursion, parallel processing, exception handling or interrupt handling. It is doubtful if the existing control primitives can be used to build new control structures. (pp. 2-5, Section 5)

Major changes will have to be made in the existing control structures of the language to meet this requirement. Capabilities for recursion, parallel processing, interrupt handling and exception handling will have to be added. Furthermore, construction of complex control structures using existing control primitives requires that a new set of control primitives be developed which allows such complex control structures to be built via mechanisms such as concatenation, procedure calls, etc. All existing implementations will have a major impact on them as a result of these changes in the language.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.P+

The language provides for GOTO and specifies that the transfers of control be limited to labels declared in the same procedures or functions in which the GOTO occurs. However it does permit branches from a main program to the labels within its procedures or functions via procedure or function invocations and label parameters. Thus it only partially meets this requirement. (Section 5.4.1)

The language should exclude transfers of control via labels from a main program to references into its procedures and functions.

Implementation of this change will have a minimal impact on existing compilers.

63. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.pt

Based on Boolean expressionT
Based on type from discriminated unionF
Based on computed choice among labeled alternativesI
All alternative must be accounted forF
Simple mechanisms will be supplied for common casesT

The language provides for IF-THEN-ELSE and SWITCH conditional control structures. In the IF statement, the Boolean expression (bit formula) has to be evaluated before the next course of action is determined (Section 5.5). However, it does not permit selection of an alternative based upon the value from a discriminated union. The SWITCH statement permits a choice from among the labeled variables. The otherwise clause is not provided for in the J3P SWITCH statement, hence all alternatives are not provided for in this case.

To fully meet this requirement the language has to be modified to include the OTHERWISE clause in the SWITCH statement. Furthermore it should allow selection of an alternative based upon the subtype of a value from a discriminated union in the SWITCH statement.

Minimal changes to existing compilers will be required to accommodate these language modifications.

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).pt

Termination can occur anywhere in the loopT
Multiple termination predicates are possibleT

Entry permitted only at the loop headT
Simple cases are clear and efficientT
Control variable is local to the loopT
Control value is efficiently available after terminationP

The FOR loop in J3B very nearly meets this requirement. It allows termination of a loop via a GOT0 or IF statement anywhere in the loop. Entry to the loop is permitted only at the top. The control variable is local to the loop and can be assigned to or used in formulas within the loop. But its value upon exit from the loop is dependent on the implementation and is not language defined. (Section 5.6)

The language should require that the value of the control variable should not only be available upon termination of the loop after a normal or abnormal exit, but should also be well defined. It must not be left to be implementation dependent.

Minimal changes to some of the existing implementations will be required to include this language modification.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.F

No recursive procedures within recursive proceduresF
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)F

The language does not permit recursive options either during procedure definition or during data definition.

The language must provide the options to define recursive procedures. If the option is not specified the procedure should be taken to be non-recursive. Thus no penalty during code generation will be paid for this option. The maximum depth of allowable recursion should also be specified in the language to keep the generated code and stacks to manageable size.

Moderate changes to existing compilers will be necessary to implement this feature in the J3P language.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.F

Able to create and terminate parallel processesF
Process can gain exclusive use of resourcesF
No parallel routines within recursive routinesF
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF
(Access rules are enforced)F

No tasking or parallel processing features are present in J3B.

Implementation of parallel processing features will require definition of language features for scheduling parallel tasks, providing for intercommunication between parallel tasks, interrupt handling, waiting and/or synchronization, program swapping, priority handling etc. This is a major undertaking which will require significant additions to existing compilers.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.F

Program can get control for any exceptionF
Parameters can be passedF
Can get out of any level of a nest of controlF
Can handle the exception at any level of controlF

No provision is available via the language features to provide the user with the facility to recover from arithmetic overflow, space overflow, hardware interrupts or any other exception situations.

Language facilities to provide for various exception situations should be added. Implementation of these facilities will involve a moderate to large effort in the existing compilers depending upon the number and type of exception features involved.

G8. There will be source language features which permit delay on any control path until some specified time or

situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

.....F

Priority specificationF
Synchronization via wait/enable operationsF
Wait for end of real time intervalF
Wait for end of simulated time intervalF
Wait for hardware interruptF
(Can enable and disable interrupts)F

The language does not permit priority specification by task or programs, wait option or any other real-time features.

Language definition will have to be extended to provide for various real-time features specified in the Tinman requirement. It will require moderate effort to implement these features on existing compilers.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....P+

Free format with statement terminator	T
Mnemonic identifiers possible	T
Based on conventional forms	T
Simple grammar	T
No special case notations	T
No abbreviations of identifiers or keywords	T
Unambiguous grammar	P

J3P very nearly meets this requirement. It allows statements to be written in free format in columns 1-72 of a line or card, terminated by a semicolon. It allows identifiers to be two or more alphanumeric characters in length. The user can thus select mnemonic names. The compiler is obligated only to examine the first eight characters of identifiers to determine uniqueness. Many conventional notations, including the left to right precedence rules, are observed by the language. There are no language features which cater only to special cases, and no abbreviations of keywords are allowed. In most cases the language constructs are clear and unambiguous. However there are some exceptions. For instance the constructs INTGR and INTR, DEF and DEFINE are syntactically very close to each other and ambiguous.

All language primitives should be syntactically distinct from each other to avoid ambiguity. Minimal changes to the language and implementations will be required to meet this requirement.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.T

J3P has well-defined fixed syntax. The user is not allowed to modify any primitives or language constructs (e.g., statements).

No conflict with other language features.

H3. The syntax of source language programs will be corosable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

....F

J32 character set consists of 26 upper case alphabets, the decimal digits, and other 18 special characters. Furthermore it allows an undefined number of implementation defined legal characters. If the implementation permits characters other than those available in ASCII, then the language permits such characters in its definition. Hence it does not meet this requirement.

J32 definition should forbid implementation defined characters, not available in ASCII character set, to be included in the language definition.

Minimal changes will be required in some implementations to provide for this language modification.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

....P

Break character existsF
(Literals are self-identifying as to type)T
(Bit-string literals for any type)P

The language provides for formation rules for literals and identifiers (Sections 2.2.3, 2.2). However, the language does not define a break character for use within the identifiers.

The language should recognize a break character (e.g., underscore or space) for use within identifiers to provide readability in these constructs.

Minimal changes in syntax checking procedures will be required to accommodate this change in the language.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

....F

J3P provides for a continuous stream of inputs. After column 72 of a card or line the first column of the next card or line is taken to be the next character in continuation. It therefore permits continuation of lexical units across lines. (pp. 2-3)

To meet this requirement the language definition will have to forbid the continuation of lexical units across lines.

Minimal changes will be required to implement this change in lexical and syntax analysis phases of compilers.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

....T

The language allows only 53 reserved words which cannot be used as identifiers.

No conflict with other requirements.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

....P

Uniform comment conventionP
Look different from codeT
Bracketed by one or two charactersT
Can contain any charactersP
Can appear anywhere reasonableP
Terminated by the end of the lineF
Compatible with automatic reformattingT

The J3P language permits two comment conventions: The comments can either be enclosed between double quotes or they can be enclosed between % signs. The user can specify anything in the comments including source code which will be ignored. Semi-colons are not permitted in comments but all other characters are. Comments are permitted anywhere in the

program except in define declarations (Section 2.2.4). Comments are not terminated by the end of line; instead they are continued over the line and terminated by either the closing double quotes or the % sign. The comment conventions do not prohibit automatic reformatting of programs.

The language will have to drop one of the two sets of marks to provide for one single uniform comment convention. It should permit inclusion of all language defined characters including the semi-colon. The comments should be allowed to appear at a suitable place within the define declarations. Comments should be allowed to be terminated by the end of line.

Implementation of these changes in the comment conventions of the language will require minimal changes in existing implementations in the area of lexical analysis.

H8. The language will not permit unmatched parentheses of any kind.T

J3P requires matching, opening and closing parentheses as well as BEGINS and ENDS.

No conflict with other requirements.

H9. There will be a uniform referent notation.PT

J3P syntax for array and table reference is identical to that of its internal function calls. They all utilize parentheses after the array, table, or function name (Sections 9, 4.3 and 4.4). But whereas the arrays are permitted to appear on either the left hand or the right hand side of an assignment statement, functions are restricted only on the right hand side of the assignment statement.

However, the language also permits pseudo variables, which can appear on the left hand side of an assignment statement. (pp. 9-1)

To truly meet this requirement the language must require that all functions be capable of use as a pseudo variable and capable of being put on the left hand side of the assignment statement.

The implementation of these changes will require moderate amount of changes in the design of library functions, procedures and assignment statements.

H10. No language defined symbols appearing in the same context will have essentially different meanings.P

In general, language symbols have unique usage, but there are some exceptions. For example, the = sign is used both for assignment and as a test for equality in conditional expressions. (Sections 5.2 and 5.5)

A distinct sign for the assignment operation should be included in the language.

Minimal changes will be required to implement this change in existing compilers.

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

....P

Most defaults affecting program logic are specified in the reference manual with few exceptions. For example, the SWITCH statement does not contain the OTHERWISE clause (Section 5.4.3). Hence the implementors decide where the control will be transferred if the value of the index exceeds the number of labels present in the SWITCH. Similarly, since no exception control statements are provided in the language, the implementation determines what to do or where to transfer the control after an overflow or a hardware interrupt.

The SWITCH statement should be modified to provide for OTHERWISE or OUT clause. Exception control statements should be added to the language to allow the user to control the flow of program in the event of an interrupt. A moderate amount of effort will be required to extend the existing implementations to include these capabilities.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

....P

Defaults specified for don't care casesT
Programmer can override the defaultsF

The language provides for a large number of defaults to be handled by implementations. Some of these are don't care type of defaults e.g., representation of data, treating a routine as CLOSED if INLIN option is not specified, and treating a procedure as non-reentrant if the user does not specify the RENT option. Other defaults, such as the ones listed in I1 are not "don't care" type of defaults. The programmer does not have the facility to override those types of defaults.

For the scope of the needed modifications, see I1.

I3. The user will be able to associate compile time variables with programs. These will include variables which

specify the object computer model and other aspects of the object machine configuration.F

The language does not provide for selective compilation depending upon the value of certain control variables, as specified in the requirement.

To fully provide for conditional compilation facility, the user should be able to specify the name and configuration of object computer including its memory size and special hardware features. These should be taken into account during the code generation process by the compiler.

Moderate changes, including where to specify these variables in the programs, their syntax etc., and the type of code to be generated for each option will have to be made in the existing compilers.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.T

J3P permits conditional compilation using conditional control structures. The syntax of IF statement in J3P is as follows:

IF bit formula; statement [ELSE statement]; .

If bit formula is a bit constant or a bit constant relational whose value is not all zero, then the statement following the semicolon is compiled, otherwise not. If an ELSE is present, only one of the statements preceding and following the ELSE is compiled. The statement preceding the ELSE is compiled when the bit constant or bit constant relational value is not all zero bits; otherwise the statement following the ELSE is compiled. (pp. 5-20)

Not compiling a statement means that syntax checking is performed, but code is not generated for that statement.

No conflict with other requirements. This facility can be extended to SWITCH statement also for constant values of the index.

15. The source language will contain a single clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

....P+

The JSP language specifications represent the kernel of the language. There are no two features which duplicate each other's capabilities with the possible exception of GLOBAL and COMPOOL scope definitions for variables. A variable can be either placed in a COMPOOL or have the GLOBAL scope, and both options will have the same effect. Hence the language nearly meets this requirement.

A redefinition of capabilities of COMPOOL and GLOBAL scopes can eliminate redundancies in the language.

Minimal changes will be required to implement these modifications in the existing compilers.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be explicitly in the language definition.

....P

The language specifications do not specify the maximum level of nesting in loops, the maximum number of variables that can be defined, the maximum number of subroutines allowed, number of nested parentheses levels in expressions, etc. It does, however, put a limit on the size of the variable name (" alphanumeric characters), number of array dimensions (3) etc. (Sections 2.2, 4.4). Thus it partially meets this requirement.

The specifications should set upper limits on several factors listed above rather than leaving them implementation dependent. These limits should be reasonably high.

Implementation of these language specified limits will require minimal changes in existing compilers.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....T

J3F language does not contain features which are dependent upon object time environment (e.g., core size, number of peripheral devices, etc.)

No conflict with other requirements.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.T

No efficiency cost for unused featuresT
Efficient code can be produced for all featuresT

The language design is such that there is no overhead in terms of generated code because of unused features. For example, there is no overhead for recursion if it is not specified by the user, no overhead for the INLINE option, etc.

No conflict with other requirements.

J2. Any optimizations performed by the translator will not change the effect of the program.T

The J3F specifications do not specify if the translator should perform optimization and how. However, the intent of the requirement is met in language specifications. Certain language features, such as INLINE, conditional IF for compilation, etc. do permit optimization of code or time and it is intended that translators should not change their effort.

No conflict with other requirements.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.Pt

J3F requires that machine code be encapsulated inside procedures which can then be called from J3F procedures or programs via normal procedure invocation methods. Machine code cannot be interspersed inline with J3F code. However, the language does not require the assembly language procedures to have a preamble specifying such details as the name of the computer, computer word size, etc. This can, however, be accomplished voluntarily by the programmer via appropriate comments.

A mandatory procedure should be established requiring machine language procedures to specify the name, size, word size and other

details of the hardware environment. This will provide readability of machine dependent routines and make their encapsulation complete.

Minimal changes in existing implementation will be required to incorporate this language modification. The entire preamble can be treated like a comment.

J4. It will be possible within the source language to specify the object presentation of composite data structures. able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.P

Encapsulated specification of representation possibleP
Space/time tradeoff can be specifiedP

J3B allows a data packing factor to be specified in its table declarations which permits a user to have no packing (N), medium packing (M) or dense packing (D) of data (Section 4.3). However, no other explicit option is available to provide the translator with the specifics of data representation.

Certain other language features allow the user to control the space/time trade off. These include INLINE option (which can save time at the expense of space), RFNT option in procedures (not specifying this option will save core space by generating non reentrant code) and IF conditional compile (which saves both time and space). However, no language defined directive is available which instructs the translator to consistently generate code to save time or space whenever and wherever possible throughout the process of compilation.

Language defined directives to the translator should be introduced which can specify certain formats for data before it is transferred to an I/O device or another computer. These directives should also specify whether the user wishes to have code generated which will attempt to save execution time or core space wherever possible.

Implementation of these capabilities will require modifications to existing compilers to accept language specified directives, in the area of I/O to be able to accept or transmit data in specified formats and in the areas of code optimization.

J5. The programmer will be able to specify whether calls on

a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.T

Open/closed properties can be specifiedT
Open and closed versions have the same semanticsT

The INLINE attribute permits a library procedure to be included in the user code at the point of invocation. If this option is not specified then the procedures are treated as closed.

No conflict with other requirements.

Extraneous Features

We recommend that the following features of JOVIAL J3F, not required by the Tinman, be kept:

- * The use of bit strings.
- * Compiler directives for table packing (none, medium, or dense).
- * LIKE table definition. This same effect could be obtained by other means if JSR were extended to include a type definition facility.
- * certain pseudo variables (functions) which can be used on the left hand side of an assignment statement. For example, the NEXT functional modifier.

We recommend that the following features of JOVIAL J3E, not required by the Tinman, be deleted:

- * Double precision floating point. This capability would be assumed by the precision specification in any extension of the language along the Tinman lines.
- * The NWDSEN functional modifier, which returns the size of a table in words. This is too hardware dependent.
- * OVERLAY, a form of free union.
- * Byte type data. This is too hardware dependent.

SUMMARY

J3P is a small language with many modern, useful features. Clarity and consistency of language feature has been emphasized throughout the language definition. The language provides for four levels of scopes (SYSTEM, COMMON, GLOBAL, and LOCAL) and specifies rules for resolution of conflicts related to levels of scopes of variables, data, procedures, etc. A COPY option is provided as a compile time facility to insert program text from external sources. The language also emphasizes compile-time checks, whenever and wherever possible, thus providing for debugging in early stages of program development. It requires type checking on procedure parameters and distinguishes between input and output parameters. It requires evaluation of constant expressions at compile time. Declaration for all variables, tables, records, and arrays have to be explicit and should be before use. A pointer mechanism is provided in the language. An uncommon feature in the language is conditional compilation of IF statement, in which the constant condition is evaluated at compile time and code is generated only for that portion of the condition which will be executed during execution time. A capability for specifying that a procedure is to be compiled in open form is also available (INLINE option).

However, there are several characteristics in which the language is lacking and fails to measure up to the Tinman requirements. The language does not provide for definition of new data types or new operators. It does not provide for features supporting real-time processing, parallel processing, or exception handling. No provision has been made for input/output, dynamic array allocation, truncation of results from the right instead of from the left, or explicit instead of implicit data conversion. Range and precision specifications for variables are not supported. The user does not have the facility to define procedures with a variable number of parameters. Procedures may not be recursive.

Despite the lack of many essential qualities specified by the Tinman requirements, J3P contains many fine characteristics. The existing kernel of the language is small and contains features which can be discarded, modified, or expanded, to meet the Tinman requirement. To add the missing capabilities to the language definition and to implement them are tasks of major proportions, but they have a reasonable chance of success. However, such efforts cannot guarantee a clean, consistent, and simple language. Starting from scratch would be a better approach to obtain these characteristics.

A COMPARISON OF
JOVIAL J73 (Level I)
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language JOVIAL J73 (Level I) to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, JOVIAL J73 (Level I) is considered to be defined by:

JOVIAL J73/I Specification
July, 1976
Rome Air Defense Center

Tinman contains 78 language requirements. This report compares JOVIAL J73 (Level I) to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols F, P+, and P- will often be used with requirements which are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

THE REPORT CONCLUDES WITH TWO SUMMARIES. THE FIRST IS OF THE FEATURES OF JOVIAL J73 (LEVEL I) WHICH ARE EXTRANEOUS TO TINMAN AND THE DESIRABILITY OF RETAINING EACH OF THEM. THE SECOND IS OF THE LANGUAGE AS A WHOLE AND THE DESIRABILITY OF MODIFYING IT TO BRING IT INTO LINE WITH THE TINMAN REQUIREMENTS.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....P

While J73 is typed in the sense that types are known at compile time, certain language features allow the defeat of type checking against which the compiler cannot protect. The constructs OVERLAY, based variables, and specified tables essentially implement free unions, and since there is no requirement (nor, indeed, any language artifact) to discriminate among the union types, the full intent of A1 is not satisfied in J73. (Sections 1 and 2)

The elimination of OVERLAY is trivial and has no impact on other features; the effect upon implementations is not known, but where OVERLAY is used (especially absolute overlays for system allocation), its elimination could be painful to circumvent.

Based variables could be eliminated or changed to conform to Tinman "pointer property" specifications -- decidedly expensive (see D6).

Specified tables could be eliminated, but this impacts J4, and would severely impact the extant J73/I compilers (namely, DEC-10, HRC, and SAMSO Fault-Tolerant J73's) which make heavy use of specified tables to optimize table space usage.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....P

Integer	T
Floating Point	T
Fixed Point	F
Boolean	P
Character String	T
Arrays	P
Records	P

Boolean -- There is no Boolean type as such in J73. However, the effect of Boolean is implemented through the use of type "Bit" in which

the low-order bit of a bit string is treated as "true" if a 1 and "false" if 0. Thus, for example, predicates are evaluated to type bit of size one rather than to the conventional type Boolean.

Arrays and Records -- J73 permits both arrays (called TABLEs) and records (also called TABLEs). However, while one can declare an array of records (that's exactly what a TABLE is), one cannot declare records containing arrays. Furthermore, J73 does not support the notion of prototype definitions (MODE in ALGOL 68 or "type" in PASCAL), such that "type generators" as called for in A2 are not fully supported.

The addition of records of arrays and prototype definitions to J73 is a major extension and would greatly effect existing compiler design. Since type generation is unknown to J73, existing compilers assume a fixed number of pre-defined types in the symbol table design, and generalizing type definition is therefore a fundamental and costly change. Syntactically, one can see that the addition of prototypes is somewhat clumsy owing to the fact that J73 declarations contain property descriptions that are scattered about the declaration. To some degree, parameterized defines may be used to effect prototype definitions. Records of arrays are more difficult because of the unique naming conventions of J73 and the lack of provision for field selection by qualification.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.P

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedT

Floating point precision is specified in the type declaration for individual variables by stating the mantissa size in bits. Thus,

ITEM FF F 27

declares a floating point variable requiring a minimum of 27 bits (excluding sign) to express the mantissa. The precision may be omitted, and the compiler supplies a default "single-precision" value that is target-machine dependent. (2.1.3.2)

A global floating point specification would affect only those variables with the default precision. The implementation impact is trivial.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.F

Treated as exact quantitiesN/A
Range and step size determined at compile timeN/A
Scaling handled automaticallyN/A

J73 does not support fixed point in level I. (Sections 1 and 2)

The full language J73 includes fixed point, and presumably the exclusion of fixed point from level I reflects a concern for implementation cost only. Clearly, the full language fixed-point capability could be drawn into a level I.5 subset without a question of language consistency. The impact on existing compilers may be expected to be moderate.

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)F

J73 defines the character type as a built-in type. The J73 character type is really a character string type whose size is fixed at the declaration. The admissible literal denotations for character type are drawn from a large set of graphics whose ordering is implementation-dependent. (1.7.1.2, 2.1.3.2)

The addition of enumerated character types may be seen as an extension of the J73 status list (the only form of enumeration type in the language). ASCII and EBCDIC could then be added as predefined enumerations, and conversions could be added, all at modest impact on existing implementations.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each

dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryP
Subscripts only integers or from an enumeration typeP
Subscripts will be from a contiguous rangeT

All array subscripts are fixed at compile time. Subscripts are confined to integer or status type (which are otherwise treated as integers). J73 does not allow character types as subscripts (except in an explicit type conversion to integer which is the character's internal bit representation), and since Tinman requires that character type be enumerated, J73 partially fails this requirement. (2.1.5)

Adding variable upper bounds to arrays will affect existing implementations in which data allocation is fixed at compile time -- moderate input. Adding character types as subscripts requires the implementation of character type by enumeration (see A5) and a fundamental change in the way subscripts are syntactically thought of in J73 -- they are now simply integers -- and in the way in which the enumeration type is conceived -- they are now simply mnemonic integers. The intent of A6 could be met by including in the table declaration the subscript type (much as PASCAL does) to include integer, status or character.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.P

Alternative structures for records are possibleP
Discrimination condition may be any Boolean expressionP

Alternative structures are implemented through specified tables, based variables, or OVERLAY. Discrimination is entirely by convention, there are no language constructs to facilitate discrimination, and arbitrary references to memory are simply achieved. (2.1.5)

The concept of discrimination is so foreign to J73 that to modify the language to include it is essentially to define a new language. The requirement in A7 impacts the three language features (specified tables,

J0VIAL J73/1
Requirement A7

5

based variables, and OVERLAY) none of which is trivially modifiable to achieve the intent of A7.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.P

Automatically defined for any type (excert...)P
Available for individual componentsT
(Assignment and reference via functions)P

Assignment and reference are not consistently defined: Type matching is not imposed for structure assignment or parameter passing and whole arrays are not assignable (only entries of TABLEs, that is, individual instances of a structure). Assignment and reference are defined for simple items (i.e., non-record, non-array types). Assignment to union types is not possible because union type is not defined in J73.

(Reference by function is possible, except that only primitive types are definable as function "results". For example, one cannot define a function which yields a component of a record or of an array or that yields a record or an array. Functions on the left-hand-side are not definable, but certain built-in functions may be used on the left: Bit, BYTE.)

The implementation of consistent assignment and reference is an enormous change to J73, especially to permit array assignment and to add all types as function results (which we construe as essential to the notion of consistent reference). Complete consistency argues for user-definable left-hand-side functions to supplement BIT and BYTE, but mechanizing these is beyond the scope of this study.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.P

Array types may not be compared, and type matching is not imposed for record comparison such that, for example, a table entry containing a character and a real may be found equivalent to an integer. (4.4, 4.5, 4.6, 4.7)

There will be a minimal impact to include array comparisons, but a major impact to include type checking for records (although this is a problem that must, in general, be solved if J73 were to meet the most fundamental intent of Tinman).

R3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P+

Built-in for all numeric and enumeration typesT
Ordering can be inhibited when desiredF

Relational operators <, >, <=, >=, =, <> apply to all numeric types including "status" types. Unordered sets unknown in J73. (4.4, 4.5, 4.6, 4.7).

R4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.P+

AdditionT
SubtractionT
MultiplicationT
Division with real resultT
ExponentiationT
Integer and fixed point division with remainderP+
NegationT

Primitive operations use familiar notation (+, -, *, /). Division with real result occurs whenever at least one of the divide operands is real; division between integers produces an integer result. A special operator (\) yields the remainder of an integer (or real) division. (4.8)

(Note: Because fixed point is not supported in J73, there is no fixed point division with remainder.)

R5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point

numbers.

....P

Never from the left for data within range?
Never on the right for integer and fixed pointF
Implicit floating point rounding beyond precision allowedT
(Run time checks can be avoided)N/A

Assuming that "data within range" excludes intermediate results that may be out of range (how else do we get truncation?) then J73 truncates on the right by implementation convention; truncation is not discussed for arithmetic operations except for assignment to floating variables in which one has the option of rounded or truncated (presumably low-order). Right truncation is always implicit for integers.

Since truncation is not properly defined, left truncation could be added at modest implementation cost. Run-time checking could be added to correct the integer truncation problem (but with the option for disabling the checking mechanism) at moderate cost (either by inserting checking code in the object program, or by interpretive execution via debugging data generated as an amendment to the object program).

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P

Short-circuit andP
Short-circuit orP
NotT
XorT

By convention, most J73 implementations employ short-circuit mode, but not by language definition. (4.5, 4.6, 4.7)

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.P

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF-

J73 does not permit assignment or reference to arrays in any context except the passing as an actual "reference" parameter. Assignment between records is permitted with conformability defined between any two records. That is, there is no type checking, and the assignment is by bit string manipulation rather than data equivalence. (Section 4)

There will be a moderate to heavy impact to add array assignment and reference with proper type checking and to add conformability checking for records. (Also discussed under requirements P1 and B2).

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

....P

No implicit conversions	I
Explicit between integer, fixed point, and floating point	T
Explicit between fixed point scale factors	F
(Explicit between integer and Boolean)	N/A
(Explicit between integer and enumerated types)	F
(Explicit between different enumerated types)	F

Built-in functions (with specification for exception conditions) are provided for INT, FLOAT, CHAR, and BIT of types float, char, signed-integer, unsigned-integer, and bit.

There is no provision for character representation of internal numeric objects nor vice-versa.

Enumeration type in J73 is treated always as integer, thus, no conversion required or defined. (1.7.1.2)

There will be a modest impact to provide the various explicit conversions.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is

truncated.

....F

Implicit conversion between rangesF
Exception condition on integer and fixed point truncationF

. Ranges not defined.

. Truncation not noted.

Run-time checks for truncation could be added without conflict with other language features. (See B5.)

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

....F

Sending and receiving of dataF
Sending and receiving of control informationF
Dynamic device assignmentF
User exception condition controlF
Installation independenceF
(Data formatting capability)F
(Reading and writing of bit strings)F

J73 defines no I/O. (Note: A non-standard superset of Level 1 implemented for Wright-Patterson AFB Avionics Lab includes a FORTRAN I/O like capability).

Modifications to implement I/O could all be made entirely external to the language, i.e., the procedure, function, table, and compool capability well provide all that is required in B10. Note that the sending and receiving of control information is clearly so hardware and operating system dependent that it seems as if this requirement is redundant with A2; that is, the enumeration type is sufficient as an agent for the control information transfer, and language artifacts beyond this primitive capability will suffer in the attempted mapping from real hardware/systems to the language constructs.

R11. The language will provide operations on data types defined as power sets of enumeration types (see F6). These operations will include union, intersection, difference, complement, and an element predicate.

....F

Union	N/A
Intersection	N/A
Difference	N/A
Complement	N/A
Membership predicate	N/A
(Set inclusion)	N/A

Sets and power sets are not defined in J73.

J73 will accommodate the addition of sets and powersets and the required operations with small impact on existing features and implementations. Some implementation dependent upper bound on set size may be included in a realistic approach (say, w , where w is the word size in bits), and this parameter would be available as an inquiry primitive in the language.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.F

Side effects must occur in left-to-right orderF
(Embedded assignments)F

The language is equivocal on the subject. J73 specifies that "within a particular precedence level, operations are combined from left to right", while later, "The evaluation order of formulas is unspecified except that address:formulas and indices (subscripts) must be evaluated before variables with which they are associated are referenced". (4.8)

We take the latter as the governing specification.

There is no language impact in specifying left-to-right evaluation order of expressions or formulas (side effects are occasioned, not evaluated); however, existing compilers will require modification to enforce this rule.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.P+

Few precedence levelsP+
No user-defined precedence levelsI
Operands of an operation are obviousI

J73 defines 10 precedence levels. This is probably more than the "few" required by C2. However, the levels are traditional and familiar. (4.8)

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.I

(Section 4)

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.T

Note: J73 prescribes constant formulas which contain floating or character operands or which contain the exponentiation operator. (4.9)

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.F

Parameter rules consistent in all contextsT
No special operations applicable only to parametersP

In procedure and function calls, constants and simple items may be passed as actual parameters opposite tables (records/arrays) and blocks, which is inconsistent with assignment which is not defined for either tables or blocks. This feature is used to defeat type checking for parameters and implements a kind of "hidden" union as formal parameters and allows the writing of user generic procedures. (2.2.3, 3.10)

There is no impact on the existing language specification to prohibit the passing of such parameters. The impact on existing programs is unknown.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.P

Actual and formal parameters will agree in typeP
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedF

See C5 for remarks about formal parameter tables and blocks. Size and subscript range of parameter arrays are fixed by the formal parameter description. (2.2.3, 3.10).

The addition of variable size formal arrays will add moderate cost to the parameter passing mechanism but should have no adverse impact on existing mechanisms for passing fixed arrays.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P

Act as constants (call by value plus)F
Act as variables (call by reference)P
Exception controlT
Procedure parametersT
(Act as variables, but call by value)T
(Act as variables, result parameter)T

Call by reference parameters are restricted to arrays and blocks. Call by value allows assignment to a formal parameter (which does not alter actual parameter). Exception control is by label parameters. J73 also allows "value output" parameters, but these are restricted to non-composite types, i.e., to scalars, single array components, or single record components. (2.2.3, 3.10)

Reclassifying formal parameters requires a significant change to the syntax of the language; parameters are currently classified implicitly. Doing away with output parameters would require complicating the actual parameter descriptor in order to accommodate passing of record and array components. This would have a moderately heavy impact.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.PT

Above properties optionalPT
Above properties are fixed at run timeF

Generic procedures are partially achievable through the use of formal array parameters against which actual parameters of virtually any type may be passed. Thus, the intent of C9 is achieved at the cost of some clumsiness and obfuscation. (2.2.3, 3.10)

This requirement is in such blatant conflict with the strict typing conventions dictated by Tinman, that one is at a loss to suggest a scheme for meeting C9 that is not a hopeless kludge. In fact, the existing J73 mechanism is no worse a kludge than any other (e.g., ALGOL 68 'mode cuttype').

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

....F

Variable number of arguments possibleF
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeT

The number of arguments is fixed for all user-defined and built-in procedures and functions. (2.2.3, 3.10)

The scope of change to permit variable length argument lists is not accommodated within the constructs available in J73. Such implementation approaches as sublists, for example, require moderate to heavy changes in parameter passing conventions and syntax. The simplest syntactic approach would be to enclose formal sublist parameters in parentheses and to use some variant of subscript notation to select the sublist member, all of which requires a hidden parameter, accessible to the program, that identifies the length of the variable part.

D1. The user will have the ability to associate constant values of any type with identifiers.T

The J73 DEFINE permits this, as, for example:

DEFINE DOUBLE#ONE "1.0M6?".

(2.5.1)

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P

Literals for all built-in typesP
Consistent interpretation in program and dataN/A

J73 provides literals for all non-structured data, but provides no array literals nor record literals. (5.3)

No I/O is defined in J73. (See B10.)

It would be of modest cost to add structured literals to J73. The main problem will be to find some notation that doesn't provoke outrage from some corner or other.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P

Initial value can be specified as part of the declarationP
Initialization occurs at allocation scope entryP
No default initial valuesT

The syntax for data declarations includes the facility to specify initial values for so-called "RESERVE" data; data defined as allocated to its containing scope ("IN" data) may not be initialized. For arrays, initial values may be specified by particular subscript; repetition counts are also allowed. Allocation of initialized data occurs at

compile time, i.e., it is globally static regardless of scope of declaration. (2.1.4, 2.1.5, 2.1.7)

Initial values can be trivially provided for "IN" data. It would seem to be degrading of efficiency to eliminate the global static allocation, although providing dynamic initialization at scope entrance for "RESERVE" comes free if it is done for "IN" data.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....F

Numeric variable range specification mandatoryN/A
Fixed point variable step size specification mandatoryN/A
Range and step size specifications do not define a new typeN/A

J73 does not provide for range specification at all. (Section 2)

There would be moderate cost to add range specification to J73 (e.g., consider borrowing J-3 or Pascal notation). Of greater cost is the addition of range-checking code (See B5, B9).

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a continuous subsequence of any enumeration type.

....F

Ranges of an enumeration type are allowedN/A
No arbitrary restrictions on the structure of dataN/A

See D4. (Section 2)

D6. The language will provide a pointer mechanism which can

be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....P-

Recursive and network structures providedP-
Handles variable-value and structure-component connectionsP-
Pointer property is an attribute of a typed variableP
Pointer property not for constants, affects only assignmentF
Pointer property mandatory for dynamic allocationN/A
Allocation scope never wider than access scopeF
(Either the value or the pointer is modifiable)T
(Pointer mechanism handles procedures and parameters)P
(Remap and replace assignment have different syntaxes)F
(Built-in dynamic variable creation)F
(Variable equivalence classes are declarable)F

J73 provides for pointed-to things through the based variable -- items, tables, and blocks may be based. References to based variables occur either implicitly through an "address" variable defined in the declaration or explicitly through an "address:formula". The address variable or formula is nothing more than an integer expression that yields a machine address. There is no indelible association between a pointer and the pointed-to object. There is, therefore, virtually no type-checking, and, of course, even machine addressing exceptions are possible with no help from the compiler in detecting these anomalies.

Shared structures (including lists) may be mechanized by declaring record fields which by convention are used as bases or pointers to other instances of the record type. (There are other, perhaps more common and better protected ways of implementing lists in which a field of the record (a table item in J73) is set aside as an index to another instance -- say, the next in the list -- of the record type; all the space is taken from a statically allocated table whose definition is the record prototype.) This is perhaps best seen as what is essentially a machine-language mechanization of list structures (or shared records, networks, etc.) and carries with it the same level of error-propensity and unintended and/or unwise access to values of improper type. J73 does not, therefore, offer any language constructs that make clear and obvious the manipulation of shared or list-structured values. The based mechanism really offers a total escape from type-checking and permits nearly unrestricted access to the most primitive level of the target machine data.

J73 falls short of meeting the D6 requirement in that (a) pointer variables may be totally independent of the pointed-to object, (b) pointers may assume any arbitrary integer value, (c) pointer are not restricted to a property of a typed variable, and (d) the allocation scope can be wider than the access scope. (Section 2)

Because this requirement is so broad and complex, and because the requirement itself is not free of ambiguities, it is not clear how it would impact J73 were its provisions appended to the language. Somewhat superficially addressed, by simply treating the based property as an attribute and eliminating address formulas and restricting the property to tables alone, the intent of D6 could be met at modest cost. The affect of this one existing programs not known.

E1. The user of the language will be able to define new data types and operations within programs.

....P-

Even though the table declaration in J73 is quite flexible, the inability within the language to define records as types or to define records containing arrays severely limits the extensibility desired in E1.

Operations are definable through functions and procedures only. The language does not permit the definition of new infix operators or to redefine existing infix operators.

See the remarks under A2 for data definition enhancements. The addition of infix operator definitions is a minor effort if the type of operands is fixed in the definition; the effort becomes greater if operand node is used to select the proper operator. In the former case, the existing function mechanism within the compiler serves to implement the infix operator feature; in the latter case, operator selection becomes somewhat more complex, although the Tinman proscription against implicit type conversion eases the selection problem.

E2. The "use" of defined types will be indistinguishable from built-in types.

....F

Since defined types are not supported in J73, this requirement is not met.

Once defined types are allowed at all, consistent syntax is almost free. The real effort is in providing an extension to allow prototypes, unions, records of arrays, and the pointer property which are the essential elements missing from J73 to allow true type definition. (See A2.)

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

....P

Some attributes of type definition may be defaulted in J73: integer size, real precision, table lower bounds, character string size, parameter label definition, allocation class of variables. (Section 2)

It would be a minor change to require complete definition.

E4. The user will be able, within the source language, to extend existing operators to new data types.F

J73 does not accommodate defined types as required by Tinman.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.F

ConstructionF
SelectionF
PredicatesF
Type conversionsF
Operations and data can be defined togetherF

The requirement of E5 is entirely foreign to J73. In a sense, of course, any user-defined procedure may define a type and the operations for that type; but carrying this notion to its conclusion, we feel, violates the intent of Tinman and would render the language comparison useless.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.P-

EnumerationP
Cartesian products (records)P
Discriminated unionF
Powerset of an enumeration typeF

Status type is the only enumeration mechanism but does not define a new type in J73; it is treated instead as (unsigned) integer.

Records are definable by the table mechanism, but also do not define types, nor may they be used as type constructors. (Section 2)

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.P-

J73 does not define subsetting, but does allow free unions through specified tables, overlay, and based variables. (2.1.5.7, 1.5, 2.1.4, 2.1.5, 2.1.6)

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.F

InitializationF
FinalizationF
Allocation actionsF
Deallocation actionsF

See the remarks under E5.

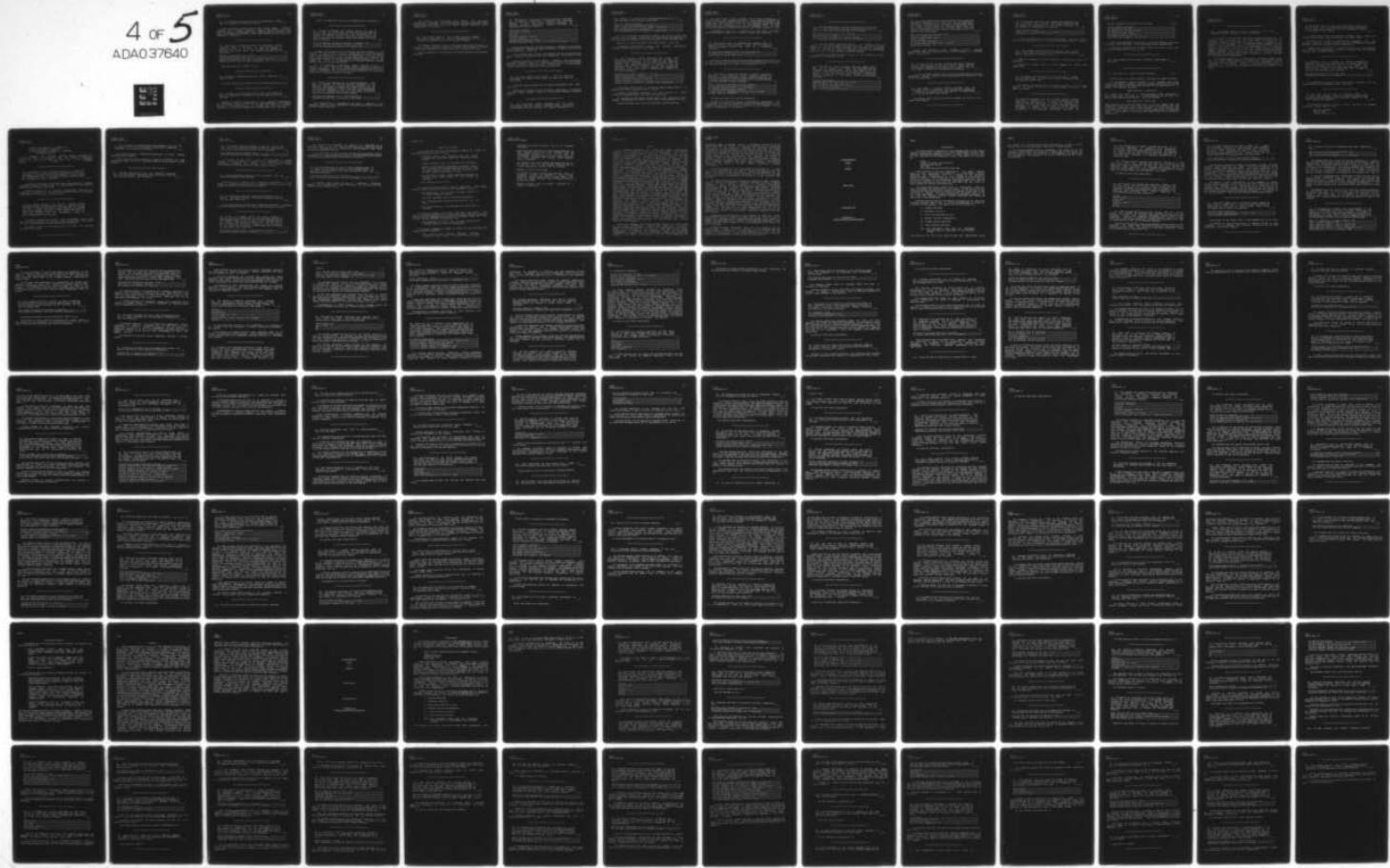
AD-A037 640 COMPUTER SCIENCES CORP FALLS CHURCH VA
DOD PROGRAM FOR SOFTWARE COMMONALITY HIGH ORDER LANGUAGE WORKIN--ETC(U)
1977 N00039-75-C-0289

UNCLASSIFIED

F/G 9/2

NL

4 OF 5
ADA037640



F1. The language will allow the user to distinguish between scope of allocation and scope of access.P

Allocation scope is never narrower than access scope. "RESERVE" variables are in effect globally allocated, but access is strictly controlled through lexical scope rules. Based variables, however, may allow an exception in permitting outer scope access to inner scope data structures. (Section 2)

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.F

Allowable operations can be limitedF
Access can be limited where usedF
External declarations need not all have the same scopeF
Naming conflicts can be avoided (renaming)F

Data types are not definable in J73.

F3. The scope of identifiers will be wholly determined at compile time.T

The scope rules of J73 conform to the F3 description (1.3.3, 2.3)

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.P

Comment: it is hard to understand F4 as a language requirement. The requirement should be that data and operations should be definable by means of a library mechanism. No language can meet the abstract requirement of supporting unspecified applications in its libraries.

(J73/I for WPAFB/AFAL supports a FORTRAN-like I/O capability.)

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

....P+

Program component libraries accessible at compile timeT
Libraries can contain foreign language routinesP
Interface requirements checkable at compile timeT

Comment: The provision in F5 to include "foreign" routines in the library is clearly not a language requirement if they are obliged to exist in an object code form that is indistinguishable from the object code of the "Common Hol", i.e., in what sense, then, is it foreign? The requirement appears to be directed at the foreign language, not to the common Hol. There is no way sensibly to evaluate existing languages against this part of the requirement.

J73 for WPAFB/AFAL accommodates library routines written in FORTRAN, in which case the J73 object code is generated to conform to FORTRAN conventions. This is an implementation dependent aspect of J73, although directives are provided to guide the compiler in generating special-form linkage to foreign languages.

F6. Libraries and Com pools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized com pools or libraries any user specified subset of which is immediately accessible from a given program.

....P

Libraries and com pools will be indistinguishableF
Immediately accessible sublibraries at any levelP

The accessibility of sublibraries is more a function of the operating system's data management characteristics than of the Hol's library/com pool properties.

J73 compools are reprocessed data, define, and subprogram definitions. Libraries are not defined by J73, but in practice, their characteristics are taken from the target machine operating system conventions. (1.6, 2.6.4)

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.P-

J73 provides access to certain machine-dependent characteristics by means of inquiry primitives (See I3). However, no mechanism is provided to peripheral equipment as special hardware. (1.2.2, 4.10.10.1)

This requirement is too abstract to provide meaningful modification estimates.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

....P

Sequential execution	T
Conditional execution	T
Iteration	T
Recursion	T
(Pseudo) parallel processing	F
Exception handling	F
Asynchronous interrupt handling	F
Control structures from a small set of simple primitives	F

J73 provides GOTO, IF, and FOR statements. Recursion is attainable using features not intended to support recursion ("Based" procedures and the DEF facility). (3.2, 3.4)

GOTO is allowed out of scope, and parameter labels are permitted as J73's solution to exception exits from procedures. This conflicts with G2.

Parallel processing is major impact. Exception and asynchronous interrupt handling could be implemented through the library somewhat trivially if certain constraints were imposed (such as prescriptions against global exits (GOTO) from exception routines).

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

....F

J73 allows direct branches out of scope and parameter label GOTOS. (3.2, 3.4)

Branches out of scope could be trivially proscribed. Eliminating parameter labels leaves a small gap in providing procedure exception exits.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on

the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.P

Based on Boolean expressionT
Based on type from discriminated unionF
Based on computed choice among labeled alternativesT
All alternative must be accounted forF
Simple mechanisms will be supplied for common casesF

J73 does not provide discriminated unions. J73 switch supplies the familiar CASE facility. Unaccounted for alternatives are undefined in J73 (in the SWITCH statement ELSE may be omitted in the familiar way).

There are no simple mechanisms for common cases. (3.7, 3.8)

Implementing discriminated unions must precede implementing computed alternatives based on union type.

64. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).P

Termination can occur anywhere in the loopP
Multiple terminating predicates are possibleP
Entry permitted only at the loop headP+
Simple cases are clear and efficientT
Control variable is local to the loopF
Control value is efficiently available after terminationF

Termination within a loop is possible by GOTO, RETURN, STOP, or by turning the "WHILE" condition false. (3.9.2)

Multiple terminating predicates are only possible by normal conditional branches within the loop that force exit.

The language permits multiple entry loops by not diagnosing them. However, essentially, J73 calls a branch into a loop undefined (3.9.2).

Control variables may be any non-composite, scalar variable.

The value of the control variable after normal termination is available only as an implementation quirk. Since ordinary variables are used as loop control variables (3.9.1), it follows that on abnormal loop termination, the variable holds its last assigned value. At normal termination, the last assigned value is not specified, i.e., the user cannot know whether incrementation precedes as follows the test. (3.9)

Restructuring loops is a moderate task and quite localized. Compilers should be obliged to diagnose branches into FOR-loops.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

....P+

No recursive procedures within recursive proceduresT
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)F

Recursive procedures are defined by based procedures, DEF PROC, and REF PROC. Since full J73 defines recursive procedures, one can see that the recursive capability in Level I is unintended. (2.2)

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

....F

Able to create and terminate parallel processesF
Process can gain exclusive use of resourcesF
No parallel routines within recursive routinesF
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF
(Access rules are enforced)F

J72 makes no mention of parallel processing.

It would be a very heavy impact to add parallel processing. This requirement is vague and potentially defines an operating system, so to evaluate the scale of the effort to include it in any language at this level of investigation is impossible.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.P-

Program can get control for any exceptionF
Parameters can be passedF
Can get out of any level of a nest of controlF
Can handle the exception at any level of controlF

J73 does not address exception handling. In practice, there is nothing in J73 that prevents exception handling. The stylized system interface is handled by machine language but J73-callable procedures. While J73 provides no exception handling artifices, implementations written in J73 do permit rational exception handling.

It would be a moderately heavy impact to add exception control constructs and discipline to J73.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.F

Priority specificationF
Synchronization via wait/enable operationsF
Wait for end of real time intervalF
Wait for end of simulated time intervalF
Wait for hardware interruptF
(Can enable and disable interrupts)F

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

....Pt

Free format with statement terminatorT
Mnemonic identifiers possibleT
Based on conventional formsT
Simple grammarPt
No special case notationsPt
No abbreviations of identifiers or keywordsPt
Unambiguous grammarT

Special case notations: status constants (2.4.1). Keywords abbreviated: PROC, type specifiers (e.g., F for FLOATING, S for INTEGER, etc.). (2.3.1, 2.1)

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

....T

J73 does not allow modifications of built-in operations in any way: precedence, operator meaning, and language words are irrevocably fixed by the J73 definition.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

....T

The special marks, digits, and letters defined for J73 (5.1) are a proper ASCII subset.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

....P+

Break character existsT
(Literals are self-identifying as to type)P+
(Bit-string literals for any type)T

J73 uses the apostrophe for the break character. The only literal that is not self-defining is the unqualified status constant. (1.3.1)

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

....F

Lines are undefined in J73; the input is a continuous stream of characters.

It would be a trivial change to the language to satisfy this requirement.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

....T

J73 defines 68 keywords which are reserved (5.2.1). 15 of these keywords are unused in the Level I subset, leaving a net of 53. This seems few enough.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not

prohibit automatic reformatting of programs.P+

Uniform comment convention	I
Look different from code	P
Bracketed by one or two characters	T
Can contain any characters	T
Can appear anywhere reasonable	T
Terminated by the end of the line	F
Compatible with automatic reformatting	I

Since "any characters" may appear in a comment, comments can indeed look quite a bit like code. (This requirement seems inconsistent).

Comments are delimited by the double quote mark which is also used to delimit define strings (5.4, 3.5.1).

H8. The language will not permit unmatched parentheses of any kind.T

H9. There will be a uniform referent notation.F

Function calls use round brackets, subscripts use square brackets.

The replacement of square brackets in all contexts with round brackets introduces a syntactic ambiguity between replication counts for preset table data and the subscript notation for the table entry to be preset. For example, currently

" TABLE AA[1:10]F = (AA*2)(0);

parses such that (AA*2) is a "number:formula" which represents a replication count (AA must be a named constant, and the count is a compile-time constant expression). Then, the program

TABLE AA[1:10]F = [AA*2] (0);

identifies the single entry at AA*2 which is to be preset with the initialized value zero. The former case presets AA*2 entries with zero beginning, by default, with the first entry). By this example, the replacement of square with round brackets demonstrates the ambiguous syntax. A new syntax for initialized data would solve this problem at modest cost.

H10. No language defined symbols appearing in the same context will have essentially different meanings.P

J73 uses the = for both the assignment and equality operators. In addition, the @ is used for both based variable reference and for reentrant procedure calls (which some might argue is consistent -- a so-called based procedure is one whose data space is supplied by the caller, i.e., the procedure data is based but not the procedure itself, which is the reason for judging it inconsistent here). In addition, subscript brackets are used for a variety of purposes whose syntactic context determines the meaning. The same is true for for unqualified status constants. Lastly, parentheses are used to delimit constant expressions, without which there would be introduced syntactic ambiguity in initialized data constructions.

The language could made more consistent at probably modest cost, but many would object loudly to the elimination of unqualified status constants, and the subscript cases merit more examination than is affordable in this study. Clearly a new assignment operator is a trivial change, and the based procedure call question is arguable in any event.

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

....P

J73 defaults the range and precision of numeric data. However, the language does provide inquiry primitives. (2.1.3.2, 1.2.2)

Similarly, the internal representation, character size, and collating sequence for the character type are implementation dependent defaults (which the programmer may not override).

Default range and precision is trivially eliminated. The character type internal representation, collating sequence, and size defaults present a more difficult problem.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

....P

Defaults specified for don't care casesF
Programmer can override the defaultsF

The programmer may override size and precision defaults, but not those relating to the character type. (See 11.)

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

....Pt

J73 provides a variety of target machine parameters as language primitives. These include (1.2.2)

Bits per character
Bits per word
Address units per word

Number of characters per (packed) word
Bit size for address formulas
Lowest and highest addresses of statically
allocated data space

In addition, the language provides inquiry functions for (4.10.10.1) size of allocated data for "based" procedures and the bit size, byte size, and word size of data structures (items, tables, blocks).

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

....P-

Conditional compilation is possible only through the use of special directives (!SKIP, !BEGIN, !END). The conditionality is not expressed through CASE or IF-like constructs.

Moderate to heavy cost to implement compile-time variables and predicates, depending of course, on the sophistication of the control constructs (are FORs allowed, for example?)

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

....T

J73 Level I is the basic subsets. (This requirement sounds like the basis for extensibility, in which sense, J73 partially fails except for that provided through procedures/libraries/compools.)

J73 Level I offers few redundancies (IF and SWITCH, for example) and these are useful.

I6. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.F

J73 leaves compiler capacities/limitations as almost totally implementation dependent.

Setting compiler limits within the language definition may have significant impact on existing implementations which are unknown at this writing.

I7. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.T

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs."

No efficiency cost for unused featuresP
Efficient code can be produced for all featuresP

The language implies certain costs for little-used or unused features (procedure calls to handle based procedures and parameter procedures), although the language does not exclude clever implementations that may circumvent these apparent costs.

J2. Any optimizations performed by the translator will not change the effect of the program.1

Insofar as it is provable that successive translations of the source to object environment yield an equivalent program, then to that degree optimization will not change the effect.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.P-

J73 does not support machine code insertions, but there is nothing to prevent procedure calls to programs written in machine language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.P+

Encapsulated specification of representation possible1
Space/time tradeoff can be specifiedP

J73 permits the programmer to allocate the components of a structure down to the bit level through the use of specified tables (2.1.5.7). Allocation can be controlled to machine address level with the OVERLAY statement (1.5.1).

Space-time tradeoffs are specified through table/table-item packing specifiers to yield unpacked/fast-access, somewhat-packed/fast-access, or tightly-packed/slow-access alternatives (2.1.5.5.2).

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

....P+

Open/closed properties can be specifiedP+
Open and closed versions have the same semanticsP

J73 defines only closed routines as procedures, functions. However, the parameterized define mechanisms (2.5.1) may be used to obtain the open routine effect.

Extranous Features

We recommend that the following features of JOVIAL J73 (Level I), not required by the Tinman, be kept:

- * Character string type (assuming that the Tinman requires only a character type, not a character string type).
- * Result parameters (output parameters) for procedures.
- * Control of the internal structure of records through (1) specifying the internal arrangement of the record as either parallel or serial and (2) specifying the compiler packing algorithm to be used.
- * The REDUCIBLE directive, which allows optimization of function calls which cannot be done without the directive.
- * The functions ABS and SIGN.

The following features provide a useful capability. They should either be kept intact or replaced by some equivalent feature:

- * Bit string type. The bit string constant appears to be overly elaborate, however.
- * The SHIFT function, used to shift bit expressions.
- * The bit-manipulating operators NOT, AND, OR, XOR, and EGV.
- * The trace directive. This debugging aid could well be expanded.

The following features are machine dependent, but useful. They should be retained because of this usefulness, but it should be required that any use be bracketed (encapsulated) by statements which mark them as being machine dependent (requirement J3):

- * The functions BIT, BITOR, BYTE, and CHAR, which permit accessing bits and bytes of expressions.

The following features of JOVIAL J73 (Level I), not required by the Tinman, should be deleted:

- * The various size function (BITSIZE, BYTESIZE, WORDSIZE), which return the number of storage units

allocated to various entities. This is too hardware dependent.

- * Based data and procedures and the related features of the language (e.g., the DSIZE function and the interference directive). This is an area of programming entirely foreign to the Tinman and, until the Tinman addresses the subject, these features should be removed.
- * The function LOC, which returns the machine address of an entity. This is too hardware dependent and its primary function is probably that of a pointer.
- * OVERLAY, a form of free union.
- * The ability to return through more than one level of procedure linkage by executing a single RETURN statement. While perhaps useful in some cases, this tends to obscure program flow and thereby drive up maintenance costs.
- * Blocking of data. This is properly a function of a loader or link editor.

Summary

As a result of evaluating J73 against the Tinman requirements, several good features of the language which satisfy or come close to satisfying these requirements have been identified. The language requires type checking of all entities at compile time, permits precision specification for individual variables, requires that the number of dimensions, type, and lower subscript bound of arrays be fixed at compile time, permits relational operators for comparing all primitive data types including enumeration types, allows various built-in arithmetic operations including integer division with remainder, permits explicit conversion between various data types through functions such as INT, FLOAT, CHAR, and BIT, and establishes a few, easy to understand rules for expression evaluation and operator precedence. Rules for handling parameters for procedures and built-in operators are consistent throughout the language definition. The language provides for exception control parameters (i.e., labels) as well as procedures as parameters. It has the capability of assigning identifiers to constants. It permits initialization of most data types at the time of their definition. It allows the definition of recursive procedures and recursive data structures with the use of a pointer mechanism. The language also requires that all items, arrays, tables, etc., be explicitly declared before use, hence that no implicit declarations will be permitted. The scope of these identifiers is also determinable at compile time. The user can have access to both compilers and libraries. The syntax of the language is consistent, allows free formatting, and does not provide for special or rare cases. The language syntax cannot be modified by the user or the implementor in any way. Single quotation is permitted as a break character for use within names, thus allowing for mnemonic names to be used as identifiers and to increase readability of programs. The language has 68 keywords, all reserved. J73 assigns a unique meaning to most symbols; the only exceptions are =, which is used to denote both assignment and logical identity, and ?, which is used for both based variable reference and for reentrant procedure calls. No implementation dependent defaults are permitted with the exception of range, precision of numeric data, the internal representation of character type, and the collating sequence. The programmer has the facility to override size and precision related defaults. J73 provides a variety of target machine related language primitives. The user can specify the bits per character, bits per word, number of characters per packed word, lowest and highest addresses of static data storage, etc. The language is designed around a level 1 kernel and provides for extensibility. It also permits user defined open and closed routines.

There are, however, many characteristics in which J73 falls short of the Tinman requirements. Primary among these are the lack of parallel processing, exception handling, and input/output capabilities. There is no facility in the language which can provide global specification of precision for arithmetic, range of variables, enumeration of unordered sets, and truncation of numbers from the right instead of the left. The language does not allow definition of

procedures with a variable number of arguments, nor does it define scalar operations for arrays. J73 does not allow the user to define new data types or new operators. Com pools and libraries are treated differently. The GO10 statement is unrestricted in its use and allows transfers of control into and out of the scope in which it appears. The loop control variable is not local to the loop and has to be defined before use; that is, outside the range of the loop. The concept of uniform referent notation is ignored by the language; the syntax of function calls is different from array subscript notation. Moreover, the functions or procedure names cannot be used on the left hand side of the assignment statement like other data variables. The language definition also fails to specify machine independent compiler limitations.

Language designers have two alternatives in modifying J73 to meet the Tinman requirements. The first is to alter, delete, or extend the language capabilities, use the existing syntax and semantics as the base, and arrive at the desired language. The second is to draw upon the strengths of J73 conceptually, and to design a new language from scratch.

The first alternative will require the addition of many features, such as tasking capabilities, exception handling by means of control features or other suitable mechanism, a complete input/output capability, ability to define new data types and operators, including the extension of new operators to built-in data types. Extensive modification of the pointer capability is required, because of the rudimentary nature of the current mechanism. In particular, options should be provided to protect the areas where pointers are pointing during dynamic allocation. Other modifications such as provision for range option in data definition, truncation of data from the right instead of the left, dynamic as well as static allocation of arrays, variable number of arguments in procedure definitions, etc., will have to be provided for. Certain other features of the language, such as automatic default substitutions for integer size, real precision, table lower bounds, etc., will have to be discarded. The overall modification process for J73 will be of major proportions, but it can be accomplished. The syntax of the language obtained by this process will certainly be clumsy at places since it contains a blending of the old with the new.

Development and definition of a new language to meet the Tinman requirements will be a major task indeed, but it has a better chance of arriving at a language which is consistent and elegant in its syntax, is modular and extensible, as is capable of efficient implementation. Many of the good and useful concepts of J73 can be included in the design of such a language.

Regardless of the alternative used, the new language will be significantly different from the existing J73 and will probably not allow existing J73 programs to be easily modified to run under the new version of the language. In such a case, there seems to be no point in claiming any relationship between the new language and J73.

A COMPARISON OF
PEARL
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language PEARL to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, PEARL is considered to be defined by:

PEARL
Subset for Avionic Applications
Language Description
June 1976
FSG Elektronik-System-Gesellschaft mbH

Tinman contains 78 language requirements. This report compares PEARL to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - Only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols P, P+, and P- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of PEARL which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring in into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....T

PEARL requires that all variable data must be defined by a data name and a data type. Constant data that are to be associated with identifiers must be similarly declared and must also contain the invariable (INV) attribute. The language also requires the types of arrays and each element of a data structure to be specified by the user. Hence the language meets this requirement. (pp. 38-47)

No conflict with other requirements.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....P

Integer	T
Floating Point	T
Fixed Point	F
Boolean	P
Character String	T
Arrays	T
Records	P

The PEARL language definition permits integer data type (called FIXED), and provides for floating point, character string, array, and structure (record) type, although structures cannot contain array components. (Arrays of structures are permitted.) There is no fixed point type. Most of the functions of Boolean are satisfied by bit type of length one, but there are no Boolean constants.

Addition of the missing types would require a modest effort in most compilers. For fixed point the hardest problem is the definition of acceptable scaling rules. The addition of structures containing arrays and a Boolean type is relatively easy.

A3. The source language will require global (to a score) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.P

Global arithmetic precision specification mandatoryU
Individual variable precision specification permittedF

Floating point variables in PEARL can be given a length attribute (LENGTH), which specifies the number of bits in the mantissa of the internal floating point representation. This gives something of an ability to specify precision, but it falls short because there is no acknowledgement of the different normalizations extant (binary, octal, hexadecimal, etc.). One assumes that this specification is the minimum number of bits in the mantissa, but the manual has no explicit statement to this effect. These specifications can be made globally and individually, but the manual does not say that either is required.

It would be a simple modification to the language to require floating point precision specifications, and only a simple modification to compilers if they are to be interpreted as a directive to the compiler when it chooses between various precisions at its disposal on the target machine. If, however, they are meant to have some impact on the encoding of comparisons and expression evaluation -- as suggested by the Tinman elsewhere -- then the cost becomes significantly higher.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.F

Treated as exact quantitiesF
Range and step size determined at compile timeF
Scaling handled automaticallyF

Fixed point, in the Tinman sense, is not supported at all by PEARL.

The addition of fixed point requires a moderate effort in most compilers. It is doubtful that treating fixed point data as exact quantities is practical, however.

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)F

The language does not provide the user the capability to enumerate new character sets, hence it does not provide any conversion capability between sets. Conversion routines between ASCII and EBCDIC are not required in the language definition, and if present in the library, are an accident of implementation. The language defines its character set to consist of 26 alphabetic characters (A-Z), 10 digits (0 through 9) and 15 special characters. Furthermore, it allows installation dependent special characters to be added to the character set. (pp. 31)

To met this requirement the language must provide for some facility (e.g., Status Variables) which can allow a user to define his own character set for a given program. The library must support the conversion routines to convert the user specified character-set into the natural set for the machine on which the compiler is implemented. This should include routines which can convert ASCII code to EBCDIC and vice versa.

Implementation of this facility will affect all existing compilers. An approrriate place in the program will have to be established to define new character sets. All processing for character strings, literals, etc., shall have to be modified to provide for the new character set. Several conversion routines will have to be added to the library.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P

Number of dimensions is fixed at compile timeT
Type is fixed at compile timeT
Lower subscript bound is fixed at compile timeT
Upper subscript bound is fixed at scope entryF
Subscripts only integers or from an enumeration typeP
Subscripts will be from a contiauous rangeT

PEARL
Requirement A6

4

The user is required to specify the number of dimensions of the array at compile time as well as the type and the upper bound of the subscript. The lower subscript bound is fixed at 1. The user-provided upper subscript bound must be an integer constant. Subscripts are integers only.

To fully meet this requirement the capabilities of fixing upper subscript bounds at scope entry rather than compile time and using elements of an enumeration type as subscripts must be added to the language. Both of these would have only a moderate cost, with the cost of the latter being a part of the cost of adding enumeration types in general.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

....F

Alternative structures for records are possibleF
Discrimination condition may be any Boolean expressionF

Alternative structures are not supported by PEARL at all.

The addition of such a capability typically has a heavy impact on the syntax of any language, but beyond this the costs (e.g., to the storage allocator and code generator) are only moderate.

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.P

Automatically defined for any type (except...)P
Available for individual componentsT
(Assignment and reference via functions)F

The language defines the assignment and reference operations for numeric, bit, character, duration, clock, etc., types of data. These operations are also defined for elements of arrays and structures, but not for conformable arrays or structures. Functions can only be referenced, not assigned. (p. 71-72)

To extend assignment to conformable arrays and structures would require a minimal effort. Allowing assignment to functions would require substantially more work.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.PT

The equality operator, ==, is provided in the language for logical comparison of numeric, character, bit, or time type of data. Comparisons of arrays or records (structures) is not defined. A comparison results in a bit (not necessarily realized) being set or cleared: the value is set to '0'B if the relationship is false (not equal). (p. 66)

The addition of array and record comparison requires a minimal effort.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P

Built-in for all numeric and enumeration typesP
Ordering can be inhibited when desiredF

PEARL provides for EQ, NE, LT, GT, LE and GE relational operators for comparison of numeric data types. However, enumerated data types are not supported at all. (p. 66-67)

To meet this requirement the language must provide for status variables and extend the definition of all relational operators to the enumerated data types. Mechanisms should be provided in the language to inhibit relational operations wherever unordered sets are intended.

Implementation of these modifications will affect all existing compilers of the language. The syntax analysis, expression processing, and logical expression processing portions of the compiler will be affected to a moderate extent.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.Ft

Addition	T
Subtraction	T
Multiplication	T
Division with real result	T
Exponentiation	T
Integer and fixed point division with remainder	F
Negation	T

All the operations specified in this requirement are included in the language with the exception of integer division with remainder. (pp. 60-61)

A new operator for integer division with remainder needs to be defined and included in the language. Minimal changes will be required in the language syntax and expression processing routines to implement this feature.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point

numbers.P

Never from the left for data within rangeU
Never on the right for integer and fixed pointU
Implicit floating point rounding beyond precision allowedU
(Run time checks can be avoided)U

The language does not specify rules for truncation. It is possible to declare the length, in bits, of integer quantities, so it appears that truncation on the left is possible. Since no language defined constructs are provided for range or precision specification, no rules are specified for rounding beyond user specified precision. All details of truncation and rounding are implementation dependent.

To meet this requirement the language will first have to provide for RANGE and PRECISION specification. Then language will have to specify rules that truncation should take place only from the right. For intermediate computations, rounding for floating point numbers should take place beyond the precision specified by the user.

Implementation of these features in the language syntax and expression processing routines will require moderate changes.

B6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P

Short-circuit andP
Short-circuit orP
NotT
XorF

The language provides for AND, OR and NOT Boolean operators but does not provide for XOR. The language also does not explicitly specify that the evaluation of expressions containing AND and OR should be done in short circuit mode, nor does it forbid it. Hence the evaluation of these operations is implementation dependent. (p. 65)

The Boolean operations should include an XOR operator. The definition of the AND and OR operations should explicitly emphasize that short circuit mode is required. Some of the existing compilers which are not currently utilizing short circuit evaluation will have to be modified slightly.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

....F

Scalar operations on arraysF
Assignment between records and arrays of conformable typeF

The PEARL language definition does not allow arrays and structures to be referenced as a whole (pp. 42,45) except as procedure parameters and at initialization. This implies that scalar operations (e.g., +, -, *, /) cannot be performed on arrays as a whole and must be restricted to their individual elements. A similar interpretation forbids assignment operation to be performed between conformable arrays or records as a whole.

To meet this requirement the language must extend the definition of scalar operations and assignment to conformable arrays and structures as a whole. This may require restriction of numeric operators to only the numeric fields. It may also require special definitions for * and / operators for multidimensional arrays (e.g., the * operation for two-dimensional arrays should be different from matrix multiplication).

Implementation of extended definitions of these operators will require minimal effort in the existing compilers.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

....P

No implicit conversionsF
Explicit between integer, fixed point, and floating pointT
Explicit between fixed point scale factorsF
(Explicit between integer and Boolean)T
(Explicit between integer and enumerated types)F
(Explicit between different enumerated types)F

The language allows implicit conversions during assignment operations if the mode of the expression on the right is different from the mode of the symbol on the left. These conversions include automatic conversions from floating to fixed (integer), floating to bit, and

adjustments for variables of different bit and character lengths (pp. 71-72). Operators for explicit conversions (e.g., FLOAT, FIX, CHAR, EIT, FIT) are also provided (p. 69). Fixed point is not supported at all.

The language definition should forbid all implicit conversions, including the ones for assignment statement. Built-in operators should be provided in the language for conversion between fixed point scale factors at the time a fixed-point capability is added. These modifications to the language definition and the existing compilers will require a minimal amount of effort.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

....F

Implicit conversion between rangesN/A
Exception condition on integer and fixed point truncationF

Since the language does not permit user specification of ranges, the question of conversion between ranges does not arise. The language also does not require that an exception condition be raised when an integer or fixed point quantity is truncated during computations. These aspects are implementation dependent by the language definition.

To meet this requirement the language definition has to permit user specification of ranges. Then it should allow automatic conversion between these numerical ranges. The language definition should also require a run time exception condition to be raised if an integer or fixed point value is truncated.

Minimal changes to the language syntax and existing implementations will be required to implement these modifications. The code generator, the syntax checker and the expression processing routine will be affected.

B10. The base language will provide operations allowing programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not

be installation dependent.

....T

Sending and receiving of dataT
Sending and receiving of control informationT
Dynamic device assignmentT
User exception condition controlT
Installation independenceT
(Data formatting capability)T
(Reading and writing of bit strings)T

The PEARL language provides excellent I/O facilities. These facilities have to be specified by the user in the system division and the problem division. In the system division the user specifies the standard peripherals, the process peripherals, the sensors, effectors, interrupts, signals, and the device connections. Accessing and transmitting of data, opening and closing of files, using INTERRUPT, FND-OF-FILE, and SIGNAL-IDENTIFIER handling, etc., are done in the problem division. These two divisions combined allow the user to open or close files (using OPEN and CLOSE), declare devices (e.g., DCL (SWITCH, LAMP) DEVICE SINK BIT(1) GLOBAL;) and interrupts (e.g., DCLIR INTERRUPT GLOBAL;), connect I/O channels to devices, create, delete, access, and protect files (using SIGNAL-IDENTIFIERS), send or receive data directly from devices (e.g., TAKE, SEND etc), format and defomat data, and monitor graphic I/O. The user has the ability to connect any channel to any device, thus making his program largely installation independent. Only portions of the system division need to be changed when a computer or installation is changed. (pp. 100-131)

NO conflict with other requirements.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

....F

UnionF
IntersectionF
DifferenceF
ComplementF
Membership predicateF
(Set inclusion)F

The PEARL language does not support any user-defined data types and hence does not permit the powerset operations listed in this requirement.

Definition of these powerset capabilities is well understood and implementation will require a relatively small effort.

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.F

Side effects must occur in left-to-right orderF
(Embedded assignments)F

The language manual makes no statement about the order of evaluation of operands.

Function references are the only PEARL item capable of having side effects. To impose a left-to-right order of function evaluation is usually a minor problem in any compiler.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.T

Few precedence levelsT
No user-defined precedence levelsT
Operands of an operation are obviousT

The language establishes precedence levels for binary and unary operators (pp. 60-69) for expression evaluation. It establishes certain well-understood rules for interpretation of expressions. The user does not have the authority to alter or define his own precedence levels for these operators. The language definition clearly defines the operands for each of the operations. Hence the language fully meets this requirement.

No conflict with other requirements.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.T

Wherever, in the language definition, both constants and variables are allowed, expressions are allowed to be substituted in their place.

No conflict with other requirements.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.F

The syntactic definition of PEARL forbids use of constant expressions in place of constants, especially in declarations and specifications (pp. 43, 54, 55, etc.). Only INTEGER-CONSTANT is allowed to be used while declaring a fixed point, floating point, bit, or character type variable; not any type of expression.

The language definition should be made flexible to accommodate constant expressions in place of constants in declarations and specifications.

The necessary changes in the syntax are trivial but it could be moderately expensive to implement in any compilers which do not already have at least a rudimentary form of this capability.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.T

Parameter rules consistent in all contextsT
No special operations applicable only to parametersT

PEARL satisfies this requirement fully. However, only procedures use parameters in PEARL, so the requirement is almost satisfied vacuously. The properties of parameters for subroutines (call procedures) and functions (function procedures) are identical. (pp. 91-97)

C6. Formal and actual parameters will always agree in type.

The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.P+

Actual and formal parameters will agree in typeT
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedF

The actual and formal parameters must agree in number and type. If the actual parameter is an array, the corresponding formal array must have the same number of subscripts and elements. Hence dynamic arrays are ruled out in the language definition as parameters and the size and subscript range cannot be passed as a parameter for dynamic arrays (p. 92).

To meet the Tinman requirements the language must provide dynamic arrays. This will permit passing array size and subscript ranges as parameters in procedure calls. Implementation of this feature will require a moderate amount of effort and will affect the parameter checking mechanisms, code generation, run-time storage allocation for arrays, etc., in existing compilers.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P

Act as constants (call by value plus)T
Act as variables (call by reference)T
Exception controlF
Procedure parametersF
(Act as variables, but call by value)T
(Act as variables, result parameter)F

Transfer of information between actual and formal parameters can be performed in two ways: (1) call by value and (2) call by reference. In the first case the INV attribute may be specified for the formal parameter, which causes it to behave as a constant on each reference. Exception handling parameters (e.g., labels) are not permitted in the language definition, nor is a user allowed to pass a procedure name as a parameter. Hence the language only partially meets this requirement. (pp. 93-94)

The language definition will have to be extended to permit exception control parameters (e.g., labels) to be included in the actual and formal parameters. It should permit other procedures to be included as parameters as well. Rules for procedures to be passed as parameters will have to be established (e.g., recursive calls, levels of recursion, entry points, etc.).

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

.....F

Above properties optionalF
Above properties are fixed at run timeF

Of the listed properties, PEARL parameters have only type, dimension, and format. All such properties are fixed at compile time.

The language definition will have to be modified to allow for the range, precision, and scale specification of variables and parameters at the user's option. The ability to include the array dimension and format in the formal parameter should also be included. All of these properties must be determinable at run-time.

Implementation of these modifications to the language definition will require moderate effort to change the existing compilers. These changes will affect the syntax analyzers and the code generators.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

.....F

Variable number of arguments possibleF
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeT

The language does not permit user-defined procedures to have a variable number of arguments.

The addition of such a capability would require a moderate effort, both in modifications to the language itself and to existing compilers.

D1. The user will have the ability to associate constant values of any type with identifiers.T

Variables, array names and structure names can be used to represent constant values, and can be protected against modification by the attribute INV (p. 47). An entity containing the INV attribute in its declaration must also have the INIT attribute unless the entity is a formal parameter.

No conflict with other requirements.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P-

Literals for all built-in typesT
Consistent interpretation in program and dataU

The language provides definitions and syntax for integer, floating point, bit string, character string, clock, and duration type literals. The language manual makes no explicit statement about consistency of interpretation between program constants and data. Therefore all the dangers alluded to in this requirement are present. (pp. 32-37, 127)

It is trivial to change the manual to require consistency of interpretation but, as the Tinman infers, this can cause amazingly expensive problems in compilers.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P+

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesP-

Variables, arrays and structures can be given initial values when they are declared by using the INIT attribute followed by an expression.

The value of the expression must be of the same mode as the data items being given the initial value. The values of variables occurring in the expression must be known when the initializing declaration is made. This implies that initialization can occur at allocation scope entry time.

In the cases of arrays and structures, if there are more data elements than values, then the surplus elements are initialized, by default, to the value of the last entry in the list (p. 46). No default initialization of other variables is specified in the manual. The manual emphasizes that the user must assign some value (either by assignment or initialization) to variables before using them (p. 41).

The default initialization of array and structure elements in cases where the data elements exceed the initial values in number should be forbidden. The user must be made responsible for initialization or assignment of values to variables before using them.

Trivial changes to the language definition and existing implementations will be required to meet this requirement.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....F

Numeric variable range specification mandatoryF
Fixed point variable step size specification mandatoryF
Range and step size specifications do not define a new typeN/A

The language definition does not require nor permit explicit user specification of ranges for numeric variables, nor does it provide a facility for step size specification for fixed point numbers. The language completely fails to meet this requirement.

To meet this requirement PEARL will have to modify the language definition to allow user specification of the range of variables at the time of declarations. It must also require step size specification for fixed point numbers.

Moderate changes to existing implementations are required to accommodate these language modifications.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.F

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataF

PEARL does not meet the first part of the requirement because it does not permit user specification of ranges. It does not meet the second part of the requirement either since it does not allow arrays to be part of structures and vice versa.

To meet this requirement the language must permit the range of values which can be associated with a variable, array, or structure to be any built-in or user defined type. Furthermore, the user should be able to make arrays a part of structures and vice versa.

All existing implementations will have a major impact to incorporate these language modifications. The ranges and the user defined data types have to be included. Then necessary modifications have to be made in the structure declarations, array declarations, syntax analyzer, storage allocation, and code generation portions of the compiler to incorporate these features.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.F

Recursive and network structures providedF
Handles variable-value and structure-component connectionsF
Pointer property is an attribute of a typed variableF
Pointer property not for constants, affects only assignmentF
Pointer property mandatory for dynamic allocationF
Allocation scope never wider than access scopeF
(Either the value or the pointer is modifiable)F
(Pointer mechanism handles procedures and parameters)F
(Remap and replace assignment have different syntaxes)F
(Built-in dynamic variable creation)F
(Variable equivalence classes are declarable)F

PEARL has no pointer mechanism at all. Hence, the language fails to meet this requirement completely.

The language definition will have to be modified to include a pointer mechanism which is capable of being associated with each type of variable, can be modified and permits modification of the variable or structure pointed to, is capable of being assigned to, and which does not permit access from outside the scope of its allocation.

Implementation of these language features will require a moderate to heavy effort on existing compilers in the areas of data types, storage allocation, assignment statement processing, data definition, etc.

E1. The user of the language will be able to define new data types and operations within programs.

....F

No facilities are available in PEARL to allow the user to define new data types or new operations.

Modification to language definition and syntax is required to allow user definition of new data types and operations. These operations can be in the form of procedures, but the language must provide the type checking facilities to these defined operations and data types.

Major modifications to existing compilers will be required encompassing the lexical and syntax analysis routines, the expression processing routines, and the type checking routines, to implement these language features.

F2. The "use" of defined types will be indistinguishable from built-in types.

....F

The language does not provide for user-defined data types and hence does not meet this requirement.

The syntax of user defined data type and operations as well as their usage should be treated the same way as built-in data types and operations are treated in the language. This will allow uniformity throughout the language and permit the difference between the built-in types and defined types to grow dim in the eyes of the users.

All existing compilers for the language will be affected to provide for this uniformity of built-in and defined types. These features will require minimal effort assuming they are implemented with those required to implement requirement E1.

F3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

....P+

The language requires that all variables, arrays, structures, I/O facilities (e.g., devices, channels, etc.) be declared in the program or procedures before use. For global definition the language also requires these entities to be specified in the procedures in which they are being used after having been declared in the main program.

Although definitions may occur in all areas of a program (i.e., module, task, procedure, and block), certain definitions are only permitted in certain areas (p. 38). For instance, a task, procedure, device, file, semaphore, interrupt, or signal definition can only occur in a module, while a BEGIN block can only be defined in a task, procedure, or another BEGIN block.

There are a small number of pre-defined mathematical functions for which no specification is necessary.

To require that the pre-defined functions be specified before use is a trivial problem, if that is truly wanted.

E4. The user will be able, within the source language, to extend existing operators to new data types.f

Because definition of new types is impossible, this facility is currently not available in the language.

While defining the facilities of user-defined data types and operations (see requirement E1), the language definition should also permit the use of existing built-in operators with user-defined data.

Assuming that this facility will be implemented in conjunction with the facilities specified in requirement E1, a moderate effort will be required to include it in existing compilers.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.f

Constructionf
Selectionf
Predicatesf
Type conversionsf
Operations and data can be defined togetherf

The language does not have any facility for defining new data types.

As part of the facility for defining new data types and operations specified in requirement E1, the language definition should also require that the user specify the operations permissible on those data. The defined types should not automatically inherit the operations of their constituent built-in types. The definable operations will include constructors, selectors, predicates and type conversions.

A moderate effort will be required to implement this feature, along with those called for in requirement F1, in the existing compilers.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

....F

EnumerationF
Cartesian products (records)F
Discriminated unionF
Powerset of an enumeration typeF

None of these mechanisms can currently be used for defining user defined types.

The language definition should be modified to include data definition through enumeration, Cartesian products, discriminated union, and powersets. Implementation of these mechanisms in the existing compilers will require moderate effort.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.T

PEARL permits no form of free union, including OVERLAY.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type

and the actions to be taken at the time of allocation and deallocation of variables of that type.F

InitializationF
FinalizationF
Allocation actionsF
Deallocation actionsF

The current definition of the language does not meet this requirement, because it has no facility for defining new data types.

While defining the new data types the language should provide for initialization of these data, and user specification of actions to be taken for allocation, deallocation and at termination.

Moderate effort will be required to implement these facilities at the same time as those called for in requirements E1, E4 and E5.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.T

The language defines global, local, and block allocation scopes for variables, arrays and structures. The system division allows the user another means for specifying allocation scopes for process signals and peripherals. The access scope for these variables is either equal to or less than their allocation scope. For example, if a variable is defined both at the global and local level, the local definition prevails and the global variable becomes inaccessible in that scope. Since the language does not allow pointers, it is not possible from an outer scope to access variables which were allocated in an inner scope. Hence the language meets this requirement.

NO conflict with other requirements.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.P-

Allowable operations can be limitedF
Access can be limited where usedF
External declarations need not all have the same scopeF
Naming conflicts can be avoided (renaming)T

Since the language does not allow user definition of new data types, it does not support the first part of this requirement. However, the renaming capability is implemented in the language through the IDENT attribute. Thus the language only partially meets this requirement.

The language must be modified to support user definition of new data types, and it must restrict the operations permissible on these types so that changes to the defined types do not alter the meaning or effect of the calling program.

This modification to the compiler will require minimal effort when implemented along with the features called for in requirements F4 and E5.

F3. The scope of identifiers will be wholly determined at

compile time.

....T

The language permits four levels of scope: System, global, local, and block (pp. 52, 75, 101). These must be specified by the user at compile time. Each identifier will have only one definition in each scope.

No conflict with other requirements.

....T

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

....T

The language supports the notion of libraries rather uniquely. It permits specification of library names for variables, arrays, procedures, and other entities through the GLOBAL-QUALIFIERS option (pp. 52, 88). It also requires that compiler libraries be provided to contain certain standard functions (pp. 92). Other routines related to applications can also be stored using any one of these options.

No conflict with other requirements.

....P

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

....P

Program component libraries accessible at compile timeT
Libraries can contain foreign language routinesF
Interface requirements checkable at compile timeT

The language allows program libraries to be available at compile time. These libraries are capable of holding various entities that are definable in the language (e.g., data, variables, procedures, and tasks). However, the language manual does not define interface with routines compiled from other languages, including assembly language routines.

The interfaces with procedures written in languages other than PEARL should be defined. This may require processing of header information in the compiled routines.

A minimal effort will be required to incorporate these features in existing compilers which check the interfaces already, but to add such a capability to an existing compiler could be quite expensive.

F6. Libraries and Com pools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized com pools or libraries any user specified subset of which is immediately accessible from a given program.T

Libraries and com pools will be indistinguishableT
Immediately accessible sublibraries at any levelT

Since the GLOBAL-QUALIFIER allows any data definition, including procedures, tasks, and data items, to be stored in the library, it appears to perform the functions of both com pools and libraries. Furthermore, selecting suitable names for sub-libraries and storing project oriented entities in them can help create project related libraries.

No conflict with other requirements.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.T

The language provides facilities for interfacing with the standard peripherals, the process peripherals, the sensors, and other hardware devices. These facilities, included in the system division of the program, are specified by the programmer for each specific installation. The problem division, containing the user's application program, is machine independent. The system division has to be modified for different computers or installations. It allows the user to specify the CPU connections with the devices by means of the user-specified channels. The SINGLE CONNECTION, the GROUP CONNECTION, and the TRANSFER DIRECTION allows specification of how any device exchanges data with any other device.

FEARL
Requirement F7

28

No conflict with other requirements.

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.P+

Sequential execution	T
Conditional execution	T
Iteration	T
Recursion	F
(Pseudo) parallel processing	T
Exception handling	P-
Asynchronous interrupt handling	T
Control structures from a small set of simple primitives	F

The language provides for sequential execution, the IF and CASE statements for conditional execution, the REPEAT statement for iteration, several parallel processing features for creation, termination, scheduling, and synchronization of tasks. In the event-dependent scheduling of tasks, an event can be an interrupt and the language allows for WHEN-CONDITION and ENABLE, DISABLE/TRIGGER, or INDUCE options to handle these interrupts. Some exception conditions such as ON end-of-file and ON signal are provided for, although no provision is made in the language for many other exceptions such as overflow, underflow, zero divide, size, subscript range, string range, area, etc. The language does not provide for recursive procedures. Hence it only partially meets this requirement.

The language must provide for recursive procedures, preferably permitting specification of a limit on the depth of recursion. Exception handling ON conditions for various situations listed in the previous paragraph should be provided.

Implementation of these features in the existing compilers will require a moderate effort.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.P+

A GOTO statement does not permit a jump into a composed entity such as a task, procedure, BEGIN block, REPEAT statement, conditional statement, CASE statements, etc., since such entities are considered self-contained. An exit from the block-tail of a task or procedure using GOTO is also not permitted (pp. 77-78). However, exits from BEGIN blocks using GOTO are not explicitly prohibited, although the manual has no examples of such a use.

No conflict with other requirements.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.P

Based on Boolean expressionT
Based on type from discriminated unionF
Based on computed choice among labeled alternativesP
All alternative must be accounted forF
Simple mechanisms will be supplied for common casesT

The IF statement (p. 78) allows the control path be selected based upon the evaluation of a Boolean expression. The CASE statement permits a choice from among its labelled statements depending upon the computed value of an integer expression only. It also provides OUT and FIN statements for cases when the integer expression is negative or greater than the number of alternative branches, but the OUT statement is not required (p. 79). Simplified forms of these statements are also permitted. For example, an IF statement need not have ELSE clause associated with it.

The language does not support discriminated union at all.

The CASE statement must be elaborated to permit multiple branches based on values other than the value of an integer expression. As discriminated union is added to the language the IF and CASE statements should be modified to be able to use the tag field. Moderate changes will be required to meet this requirement.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).P+

Termination can occur anywhere in the loopT
Multiple termination predicates are possibleT

Entry permitted only at the loop headT
Simple cases are clear and efficientT
Control variable is local to the loopT
Control value is efficiently available after terminationF

The REPEAT statement in PEARL comes close to meeting this requirement. It allows termination of the loop at any point by means of a GOTO or IF statement, it only allows entry to the loop at the top, and it permits simplification (e.g., it permits values of 1 to be substituted for expressions following FROM and BY if the user fails to provide a value and it has an optional WHILE clause for a common terminating case). It makes the control variable local to the loop and guarantees its validity within it. The language manual also specifies (p. 21) that the value of the control variable will be greater than the final value specified by the expression following the TO clause. However, that value is inaccessible. Indeed, because the loop variable is local to the loop, its value can only be accessed outside of the loop if it is explicitly assigned to some other variable before exit.

The language definition should specify that the value of the control variable be available to the user after the termination (normal or abnormal) of the loop. Implementation of this feature is trivial, the hardest part being finding an acceptable syntax.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.F

No recursive procedures within recursive proceduresN/A
(Maximum depth of recursion can be specified)N/A
(Recursiveness must be specified)N/A

The language does not support recursion.

The recursive option must be supported by the language. The default should be no recursion so that the user does not pay the price for recursive code if he does not need it.

A moderate effort will be required to provide for appropriate code generation for recursive procedures and for related interactions. Limits on the levels of recursion permitted will keep the stacks and other problems to manageable sizes.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

.....P

Able to create and terminate parallel processes	T
Process can gain exclusive use of resources	P
No parallel routines within recursive routines	N/A
No routines within parallel routines	T
Maximum number of simultaneous instances are declarable	F
(Access rules are enforced)	F

The language allows parallel processing capabilities. It permits the creation of a task by means of task declaration, its activation by means of the ACTIVATE statement, continuation by means of the CONTINUE statement, temporary suspension by means of the SUSPEND statement, and termination by means of the TERMINATE statement (pp. 133, 139-146). The task can only be declared at the module level. That is, there can be no task declaration within a procedure declaration, or within a task declaration. Thus subtasks are not permitted. There is no provision in the language to lock out files or records to prevent simultaneous access by two or more parallel tasks, but semaphore variables are available to simulate this capability in many cases. The language also does not allow specification of the maximum number of parallel tasks. Since the language does not permit recursion, the user cannot define parallel routines within recursive routines.

To fully meet this requirement the language should allow certain facilities to temporarily lock out a file or other resources for exclusive use by a certain task. The language should also allow the user to specify the maximum number of parallel tasks permissible at any one time.

The file handling capabilities of the language should be modified to include an 'EXCLUSIVE' option to allow a task exclusive use of that file until the resource is released. A moderate amount of effort will be needed to effectively implement these features in the existing compilers.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

.....P-

Program can get control for any exception	P-
Parameters can be passed	F
Can get out of any level of a nest of control	F

Can handle the exception at any level of controlF

The language does not provide for exception handling capabilities, except for the end-of-file capabilities. No provision is made to allow the user to control the flow of the program in case of overflow, underflow, zero divide, subscript range error, size error, etc. Exception handling parameters (e.g., labels) are not permitted to be passed in procedure calls.

The entire control structure associated with 'ON' conditions for error and exception handling must be added to the language. These structures should allow the user to terminate, resume or retry operations, permit constraints in flow of control, and in certain cases should permit exceptions by default.

A moderate to significant amount of effort will be required to include these features. These will affect the existing language syntax and code generation procedures.

68. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

....Pt

Priority specificationT
Synchronization via wait/enable operationsT
Wait for end of real time intervalT
Wait for end of simulated time intervalF
Wait for hardware interruptT
(Can enable and disable interrupts)T

The language almost meets this requirement, lacking only the ability to use simulated time. The task declaration includes a specification of PRIORITY and permits a task to wait until the completion of an event or a time duration before reactivating. Time-related events can occur at specified times or after specified intervals. The language also permits ENABLE and DISABLE options to enable or disable or induce certain interrupts (pp. 133-150). Special semaphore variables are provided to allow task synchronization when used with the REQUEST and RELEASE operations.

No conflict with other requirements.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

.....P+

Free format with statement terminator	P+
Mnemonic identifiers possible	T
Based on conventional forms	T
Simple grammar	T
No special case notations	T
No abbreviations of identifiers or keywords	F
Unambiguous grammar	P+

The PEARL language permits free formatting of its statements and requires a semi-colon to terminate them. The only exception to the rule is that blanks are not allowed in the composite operators (e.g., ==, >=, ** etc). The language allows the user to specify mnemonic names although it permits the compilers to scan only the first six alphanumeric characters to establish the uniqueness of the identifiers. The language constructs are in general readable and permit a left to right scan of statements. Rules for expression evaluation, parameter passing and other salient features of the language are simple and easily understood. There are no special language constructs to support rare cases. The language, however, permits abbreviations for some of its keywords. For instance the abbreviations DVC, SPC, DCL, INIT and PRIO are permitted to represent DEVICE, SPECIFY, DECLARE, INITIAL, and PRIORITY respectively. Furthermore the language provides clear syntax with few exceptions. For example, the constructs RETURN and RETURNS are syntactically very close to each other, yet have two separate and distinct functions to perform. One is used to return the value of the function while the other is used to define the mode of the function value. Two distant syntax entities should be used to perform these functions.

Minor modifications to the language syntax are needed to fulfill this requirement. Abbreviations for keywords should be forbidden. Those keywords, such as RETURN and RETURNS, which are syntactically very similar and yet perform two different functions, should be altered and made syntactically distant.

Inclusion of these modifications in the existing compilers is trivial. It may impact the user programs more severely.

H2. The user will not be able to modify the source language

syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.T

The language definition, its features and rules do not allow the user any means to alter the language syntax. He cannot alter operator hierarchies, precedence rules, define new keywords or infix operations. The most he is allowed to change is a few special characters which are installation dependent. This may alter parts of his programs, but does not affect the language defined constructs.

No conflict with other requirements.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.P

All language constructs are composed of a character set consisting of 26 alphabetic characters (A-Z), 10 numeric characters (0-9), and 15 special characters, which are all part of the ASCII character set. The language does, however, provide for installation defined special characters, some or all of which may not be formed from the 64 character ASCII set. (p. 31)

A further constraint should be placed on the installation dependent character set requiring that no special characters that cannot be formed from the standard 64 character ASCII set should be permitted in the set. Even better would be to forbid any installation dependent characters.

Implementation of this constraint is trivial.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.P

Break character existsF
(Literals are self-identifying as to type)T
(Bit-string literals for any type)F

The language provides the formation rules for identifiers and literals (pp. 32-38). It does not, however, provide for a break character which can be used to increase the readability of the identifiers. In fact, it explicitly forbids the use of blanks and special characters for use within identifiers.

The formation rules for identifiers and literals should be modified to include the use of certain language specified break characters. This would increase the readability of language constructs. Perhaps the number of characters to establish uniqueness of the identifiers should also be increased from 6 to at least 12.

Implementation of these suggested changes to the language will affect most existing translators but its scope will be minor.

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

....F

Nowhere does the language manual specifically permit identifiers, constants, etc., to be continued across lines. However, the manual explicitly states that comments or its portions can be carried across lines, implying that the same rule is permissible to lexical entities also. (p. 37)

The manual should explicitly forbid the continuation of lexical entities across lines.

Minimal changes to existing implementations will be required to accommodate this language modification.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

....P

The keywords in the language are reserved and cannot be used as identifiers. However, the number of keywords is not small (152). As a rule these keywords are informative. (See appendix A2.3.)

Part of the reason for such a large number of keywords is that many of the keywords have abbreviations which are also treated as keywords. If the abbreviations were eliminated, as suggested in requirement H1, the list of keywords can be significantly reduced.

Minimal effort is required to implement this change.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

....P+

Uniform comment convention	T
Look different from code	T
Bracketed by one or two characters	T
Can contain any characters	T
Can appear anywhere reasonable	T
Terminated by the end of the line	F
Compatible with automatic reformatting	T

The language allows a single uniform comment convention. These comments can be easily inserted into the program text between language elements. They must be delimited by /* and */. Any single character can be included in the comment, only the */ combination is excluded since it signifies termination of the comment. The language definition, however, permits continuation of comments over several lines and does not require their termination at the end of the line. These comment conventions do not prohibit automatic reformatting of programs. (pp. 37-38)

To meet this requirement the language must require that the comment terminate at the end of line in addition to terminating at */ combination.

Minimal implementation changes are required to incorporate this language change.

H8. The language will not permit unmatched parentheses of any kind.

....T

PEARL fully meets this requirement.

H9. There will be a uniform referent notation.

....P+

Function references and array element references have similar notation in PEARL, the arguments being enclosed in parentheses. However, assignment to a function is not possible. Since functions cannot return structure values, the question of dot qualification of a function name does not arise.

To allow assignment to a function would be a moderate effort.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

....T

The language assigns unique meaning to its symbols. The usage of its symbols is unambiguous. For example the symbol = is used for assignment while == is used for logical equality comparisons. There is, however, redundancy in symbols. Both := and = are used for assignment.

The redundancy in the language operators should be eliminated by simply disallowing one symbol to be used in the language. This will increase the efficiency of the language.

Minimal implementation changes will be required in the syntax checking mechanism and the symbol table to affect this language modification.

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

....P

In many instances the defaults are language defined. For instance, if a procedure is not explicitly declared REENTRANT, the automatic default is that it will be treated like a non-reentrant procedure which will not be callable simultaneously from two or more tasks. Similarly, in the SEND or TAKE statements if the user fails to specify the SYMBOL (the location of buffer) option, the default is assumed to be the storage cell or array assigned to the I/O device. However, in many other instances the language fails to define the default condition and leaves it implementation dependent. For example, no exception handling mechanisms are provided in the language. Hence, the action taken or the flow of control after an overflow, underflow, size error, subscript range error, etc., is implementation dependent. Similarly the language does not provide lockout facilities for file control. In parallel processing situations when two or more tasks vie for the same file at the same time, what happens is undefined. The language fails to specify ways to prevent such deadlock.

The language must carefully specify a means for handling exception conditions or list the default actions that compilers must take when exception situations arise. In parallel processing also, file lockout and other mechanisms should be provided to avoid deadlocks or simultaneous access of the same file by two or more tasks when that is not desired.

Implementation of these facilities will require moderate additions to existing compilers. The syntax handling routines, task handling routines, and I/O handling routines will have to be modified. Exception handling capabilities will have to be added.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

....P

Defaults specified for don't care casesT
Programmer can override the defaultsF

The language permits a large number of defaults to be taken care of by implementations. These are "don't care" types of defaults. They

include such cases in which a procedure is treated as non-reentrant if the REENTRANT option is not provided, the size of an input record in stream type of input, etc. But other types of defaults which affect the result or the flow of control of the program are not "don't care" type of defaults. This category includes provisions for exception handling, subscript range checking, etc. The language does not allow the user the facility to override these types of defaults.

The changes necessary to give an ability to override these exceptional defaults are discussed under requirement 67.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

....I

PEARL provides the user the facility to specify the configuration characteristics of the hardware system necessary to execute his problem program. This is done by means of the system division in which the user describes the hardware configuration, the standard peripherals, the process peripherals, the device connections, the sensors, interrupts, and signals. PEARL enables the programmer to declare problem specific designators for devices and device connections which can then readily be used in the problem division of his program. Thus a meaningful declaration of the process dependent data is guaranteed. Another advantage of this facility is that whenever changes in the hardware configurations have to be made, they can be restricted only to the system division without modification to the problem division. Furthermore, the system division of a PEARL program provides for the automatic generation of a problem specific executive system.

No conflict with other requirements.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

....F

PEARL has no conditional compilation capability.

The language designer must consider providing for such conditional compilation facilities. They should also establish as to where in the program the user must specify conditional expressions which are compile time evaluable. The result of the evaluation will then decide which of the several environments that are possible for the program should be generated.

This facility can usually be implemented in most compilers at a modest cost, the cost depending primarily on the source level elaborateness of the facility. The required changes usually can be isolated in the initial phase of the compiler.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.U

The language manual does not recognize nor clearly specify the avionic subset of the PEARL language as the kernel of the overall PEARL language. The preface to the manual states that there are commonalities between 'avionic subset' and 'PEARL-Basis-Subset'. However, a list of those commonalities which may possibly constitute the kernel of the language is not available. Furthermore, to qualify as the kernel of the language, these 'commonalities' of the language must be sufficient so that their implementation alone will make the full source language capability available through extensions.

The kernel of the language which contains all the power of the language must be identified. The other language capabilities should be definable and implementable in terms of the features of the kernel.

A moderate amount of effort will be necessary to define the kernel of the language if one does not exist at present.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.P

The language specifies few limits on its features which are basically machine independent. For instance, it requires that identifiers be 6 alphanumeric characters long. However, it does not specify limits on several other language constructs. For instance, there is no language specified limit on the number of dimensions of an array, on the maximum number of REPEATs within REPEATs, IFs within IFs, the maximum number of identifiers in a procedure or program, the maximum number of statements in a procedure, maximum number of formal and actual parameters, etc. Hence, it only partially meets this requirement.

To implement this change would require analysis of user requirements and establishment of limits for various features, such as the ones listed in the previous paragraph, so that these limits will not be exceeded by any application. The effect of such standardized limitations on existing implementations which meet or exceed them will be minimal, but the effect on other implementations could be heavy.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

....T

The language does not contain restrictions which have been imposed by the hardware or object environment with the possible exception of CLOCK data type and related features which require access to a real-time clock. However, this is an essential feature to support the real-time capabilities of the language. The language does not put limits on run-time storage requirements or the type or number of peripherals or other special hardware devices.

No conflict with other requirements.

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.P

No efficiency cost for unused featuresP
Efficient code can be produced for all featuresP

There are some features of the language which may be costly in terms of run time or code even when not used. Procedure calls are one such example. There is no provision for a NORECALL option for procedures which can save execution time. The user is not allowed the option of OPEN or CLOSED procedures and is thus forced to pay for closed calls.

Modifications to language features to include OPEN, CLOSED, NORFCALL and similar other optimizing features in the language will increase the compilers' capability to generate efficient code. Implementation of these capabilities will require a moderate amount of effort in the code generation and syntax analysis portions of the compiler.

J2. Any optimizations performed by the translator will not change the effect of the program.U

This is basically a translator requirement. However, certain features of the language, such as the REENTRANT option, permit the translators to generate non-reentrant code in their absence. It permits generation of more efficient code without changing the effect of the program.

The requirement is more applicable to the translators than to the languages. Languages can, however, provide certain features (e.g., OPEN, NORECALL, PACKING of data etc.) which can permit optimization of space and/or time.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.P

The system division of PFARL provides encapsulated access to machine dependent hardware facilities. The user has control of device

management and connections. He can control I/O channels and associate them with desired peripheral devices. He can also handle certain hardware interrupts and signals. He can address these devices, interrupts, etc., in the problem division. (pp. 100-109)

However, PEARL does not specify interfaces with machine code in any way. This interface is either not defined or can be achieved by a procedure call to a machine language procedure. The language manual does not specify that description of machine characteristics for which the machine code is being introduced be present in the code.

The language manual should clearly state how the interfaces with the assembly language procedures are to be established including the details of the format for such a call.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.F

Encapsulated specification of representation possibleF
Space/time tradeoff can be specifiedF

The language does not allow packing factors to be specified with the data structures, nor does it allow the length of each field to be specified in the structure. The language does, however, permit length specifications for fixed point and floating point data. No facility in the language is provided to specify a space/time tradeoff to the compiler. (pp. 54)

The language data structure definition should allow packing of data as well as the user specification of field length. Other space/time tradeoff directives should also be included in the language which will permit the compiler to try to optimize for space or run-time wherever possible.

The existing translators will have to be modified to accept and process the language space/time related directives. They will have to be modified to provide for new data structure definitions.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.F

Open/closed properties can be specifiedF
Open and closed versions have the same semanticsF

The language does not provide this option to the user.

The procedure definition should be modified to permit OPEN or CLOSED option to be provided by the user. All existing implementations will have to be modified to include this option. The cost will be moderate to high.

Extraneous Features

We recommend that the following features of PEARL, not required by the Tinman, be kept:

- * CHAR (character string) type and the CAT (concatenation) operator (assuming that the Tinman requires only a character type, not a character string type).
- * CLOCK, DUR, DEVICE, FILE, INTERRUPT, SIGNAL, and SEMA types. Although not explicitly called for by the Tinman as types, they are needed in the PEARL realization of other Tinman requirements (e.g., parallel processing, device independence).
- * Call by value parameters.

We recommend that the following features of PEARL, not required by the Tinman, be deleted:

- * The LENGTH global specification and data attribute. These functions will be assumed by range and precision specifications in any extension of PEARL to meet the Tinman requirements.
- * The FIT operator. The function of and need for this operator (which "fits" an integer of one length to another length) are unclear, but it appears that if range specification for integer data is added to the language, as called for by the Tinman requirements, then it will not be needed because explicit conversion between ranges is unnecessary, according to requirement R8.
- * The SKIP statement. This is a high-level no-op and PEARL, unlike FORTRAN, has no need for such a statement. It could functionally be replaced by a comment.

The two shift operators (SHIFT and CSHIFT) perform a useful function, but their uses are almost always hardware dependent. Therefore any use should be bracketed (encapsulated) by statements which call attention to that dependency. This is particularly true for CSHIFT (circular shift), which is often implemented in a highly dependent fashion. (It is difficult to functionally simulate a circular shift in a datum which occupies only part of a hardware register.)

Summary

PEARL supports several of the real-time capabilities called for in the Tinman completely or partially. It provides for input/output capabilities. The user has control of channel programming, device assignment, interrupt handling, and signal handling. He can access and manipulate files. He can perform graphic input or output. The language also provides for parallel processing features. There are language constructs for multiple task creation, termination, resumption, synchronization, and continuation. The user can assign priorities to his tasks. He can establish communication between these tasks. The language also supports real-time processing. Two data types (CLOCK and DUR) are provided in the language to account for real-time and time duration. The tasking facility is combined with real-time options to start, delay, or terminate tasks at certain appointed times or after certain time durations.

In addition, the language supports many other Tinman requirements. It requires explicit declaration of type for all entities at the beginning of their scope and before their use. It provides for numeric, character, clock, duration, array, and structure data types. It allows explicit mode conversions between different data types by providing conversion operators (e.g. FIX, FLOAT, CHAR, FIT, etc.). It allows global specification of the length and mantissa for fixed point and floating point variables. It requires that the number of dimensions of arrays, their type, and their lower bound be fixed at compile time, although it permits dynamic arrays whose upper bounds can be determined at scope entry time. It allows establishment of equivalence between variables by means of the IDENT attribute. The language syntax permits differentiation between assignment and logical equivalence and provides for various logical comparison operations. Checking for type and number of actual and formal parameters is required in the language definition, as are calls by value and reference. The language allows an identifier to stand for a constant. It provides for data initialization as part of data declarations and does not permit declarations by default. Additionally, the language provides a special capability for specifying the name of a library into which the files or procedures are to be stored.

Despite these characteristics that the language possesses, there are others in which the language is deficient. It does not have a fixed point data type. It does not provide for user-defined data types or operators, nor does it allow extension of existing operators to new data types. The language also does not support the notion of defining new data types by enumeration or as Cartesian products, discriminated unions, or powersets. Another significant deficiency in the language is the absence of any kind of pointer mechanism which can enable a user to manipulate lists and recursive data types. Recursion at procedure level is also not supported in the language. Furthermore, there are no exception handling control mechanisms in the language to control subscript range error, size error, overflow, underflow, etc. No less significant is the lack of a well defined interface with routines

compiled from assembly language and other high order languages. The language also fails to provide for mechanisms to control compilations to generate code for specific hardware environments.

In addition to the major omissions listed in the previous paragraph, the language fails to meet the Tinman requirements in some other ways. It does not provide for precision and range specification of variables at the time of their definition. There is no provision for the operation of integer division with remainder. The language does not require that all truncation during compilation take place only from the least significant bits on the right. Scalar operations on compatible arrays and records as a whole are not defined. Constant expressions cannot replace constants anywhere in the program, in particular in the declarations. The language does not require that the value of the control variable in a loop be available after normal or abnormal termination of the loop. Several keywords have alternate abbreviated forms. There is no language defined break character, and the language does not provide the user the option to specify open procedures.

Although a significant amount of effort will be required to modify the language definition and syntax and to extend the capabilities of the present definition of the language to include those features in the language required by Tinman, we believe that such an effort can be successful. With the excellent I/O, tasking, real-time and hardware interface capabilities that the language currently possesses, an extended language to meet the future DOD requirements can be built. Portions of the existing syntax will have to be modified, others eliminated and some new ones added. However, the language does contain enough substance and character to provide the basis for such a modification and extension.

A COMPARISON OF
SPL/I
to
TINMAN

Final Version

31 December 1976

PREPARED BY
COMPUTER SCIENCES CORPORATION

Introduction

This report gives a comparison of the language SPL/I to the Tinman language requirements (Department of Defense Requirements for High Order Computer Programming Languages, "Tinman" - 1 March 1976, Section IV). For the purposes of this comparison, SPL/I is considered to be defined by:

SPL/I Language Reference Manual for Compiler Release
4.0
Intermetrics, Inc.
Cambridge, Mass.
July, 1976

Tinman contains 79 language requirements. This report compares SPL/I to each requirement individually. If a requirement is totally satisfied, the accompanying text is a summary of the particular mechanism used. (Occasionally no text is needed if a requirement is totally satisfied.) If a requirement is not totally satisfied, the text consists of a summary of the shortcomings and such items as the scope of the changes necessary to fully meet the requirement and the impact of these changes on existing implementations.

Each Tinman requirement begins with an introductory paragraph. These paragraphs are reproduced in this report. In many cases they are followed by several single-line summaries of features in the area of the requirement. Usually these are features which are specifically called for in the requirement. A feature enclosed in parentheses, however, is one which the reviewers thought possibly desirable, even though not called for in the requirement.

Symbols placed beside the introductory paragraph and the individual features indicate the degree to which the requirement or feature is satisfied by the language. The symbols and their meanings are:

T - Totally satisfied

P - Partially satisfied

F - Fails (not satisfied at all)

U - Unclear from the documentation

P+ - Almost totally satisfied

P- - only slightly satisfied

N/A - Not applicable (used only for individual features when the requirement is not satisfied at all)

(The symbols F, F+, and F- will often be used with requirements which

are stated in one of the forms "There will be no..." or "All...", even though only T or F are technically applicable in these cases.)

The report concludes with two summaries. The first is of the features of SPL/I which are extraneous to Tinman and the desirability of retaining each of them. The second is of the language as a whole and the desirability of modifying it to bring it into line with the Tinman requirements.

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

....T

The SPL/I word for "type" is "mode". The requirement is met (see, in particular, p. 49). The modes of formal parameters must also be given (p. 79).

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

....P+

Integer	T
Floating Point	T
Fixed Point	P-
Boolean	T
Character String	T
Arrays	T
Records	T

SPL/I has all required types except fixed point; it does have a fraction type as a partial substitute. The following additional types are provided as well: complex integer, complex fraction, complex floating point, double precision integer, signal, resources, process, bit. (See pages 49-68.)

Adding fixed-point would be of moderate difficulty. See also the comments under A5 on the character type.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

....P

Global arithmetic precision specification mandatoryF
Individual variable precision specification permittedF

The precision of floating point arithmetic and variables is implementation dependent (p. 52).

A moderate change would be necessary to meet the requirement. The change does not strongly interact with other language features, except that a consistent syntax should be designed to specify properties of variables of all types. Routines to perform the arithmetic to the necessary precision would be needed; note that SPL/I does not even have double precision floating point capabilities.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.F

Treated as exact quantitiesF
Range and step size determined at compile timeF
Scaling handled automaticallyF

SPL/I has no fixed point type.

(See the comments on A2.)

A5. Character sets will be treated as any other enumeration type.F

New sets can be defined as enumeration typesF
ASCII and EBCDIC are providedF
(Conversion capability between sets is available)F

SPL/I has only one character set, and the internal representation is implementation dependent. (p. 53)

The necessary change is not hard; nor is it trivial. SPL/I does not even have an enumeration type generation capability, so both this and the ability to define new character sets would have to be added together. Moreover, it would be desirable to implement a routine to convert a character string from one character set to another.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.P

Number of dimensions is fixed at compile time1
Type is fixed at compile timeF
Lower subscript bound is fixed at compile timeF
Upper subscript bound is fixed at scope entryF
Subscripts only integers or from an enumeration typeP
Subscripts will be from a contiguous rangeF

The lower subscript bound is fixed by the language definition; it is always 1 (p. 116). The upper subscript bound is fixed at compile time (p. 64). Subscripts may only be integers; SPL/I has no enumeration types (p. 116). The other requirements are met (see p. 64-66, 116-117).

Some of the required properties are available by means of array slice references (p. 118-121); using these to obtain variable bounds is quite inefficient, however.

Assuming that enumeration types are added in any case, the change necessary here would be moderate. Even allowing only the upper bound to be fixed at run time adds a level of indirectness for references to all but one dynamic array per scope, and interfaces to the dynamic stack allocation mechanism would have to be added.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.F

Alternative structures for records are possibleF
Discrimination condition may be any Boolean expressionF

SPL/I does not provide for alternate structures for records, which are known as "structures" (pp. 62-63).

The necessary change would be of easy to moderate difficulty. Some redesign of the case statement would be required (or at least highly desirable), to permit discrimination based on the structure -- case

Labels are currently only integers. If run-time enforcement checks were wanted, more work would be necessary, although SPL/I does require full qualification for component names (p. 122).

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array, or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.P+

Automatically defined for any type (except...)P
Available for individual componentsT
(Assignment and reference via functions)F

The intent of the requirement is fully met for all types which SPL/I has -- it has no user defined types (pp. 157-158).

However, variables of certain types cannot be assigned to in an assignment statement, for good reasons (e.g., the PROCESS type; see pp. 165-167).

The only necessary change is in the wording of the Tinman requirement. The intent cannot be to require assignment and reference for every type, for some it is not meaningful or not safe.

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.P+

The '==' and '!= operators can be used with any type. However, both operands must be of the same type. (p. 96-97)

The necessary change would be very minor.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.P+

Built-in for all numeric and enumeration typesP+
Ordering can be inhibited when desiredT

The SPL/I relational operators are defined for all numeric types; the language has no enumeration types, or unordered types to which relationals can be applied. (pp. 96-97)

The only necessary change is to add enumeration types; see F6.

B4. The built-in arithmetic operations will include:
addition, subtraction, multiplication, division (with a real
result), exponentiation, integer division (with integer or
fixed point arguments and remainder), and negation.P+

Addition	T
Subtraction	T
Multiplication	T
Division with real result	T
Exponentiation	T
Integer and fixed point division with remainder	P+
Negation	T

The remainder from an integer division is not accessible, but there
is a MOD function. (pp. 102-103, and Appendix E, pp. E-5 and E-6).

The result of a division is of the same type as the operands, so
explicit conversions are necessary to get a real result from the
division of one integer by another. This is not a violation of the
Tinman, but is rather unusual.

The necessary change is trivial.

B5. Arithmetic and assignment operations on data which are
within the range specifications of the program will never
truncate the most significant digits of a numeric quantity.
Truncation and rounding will always be on the least
significant digits and will never be implicit for integers
and fixed point numbers. Implicit rounding beyond the
specified precision will be allowed for floating point
numbers.U

Never from the left for data within range	U
Never on the right for integer and fixed point	U
Implicit floating point rounding beyond precision allowed	U
(Run time checks can be avoided)	U

Nothing is said about the ranges of results, or about truncation.

P6. The built-in Boolean operations will include "and", "or", "not", and "xor". The operations "and" and "or" on scalars will be evaluated in short circuit mode.P+

Short-circuit andP
Short-circuit orP
NotT
XorT

All the required operators are present, but AND and OR are not necessarily evaluated in short-circuit mode. (pp. 87-91)

The required (and questionable) change is of moderate difficulty. The rules for evaluation order in expressions would have to be changed.

P7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.T

Scalar operations on arraysT
Assignment between records and arrays of conformable typeT

Component by component operations on arrays are provided (pp. 91-92). They are apparently not provided for records (p. 92). Scalars are considered compatible with arrays having components of the same type (p. 102), and scalar actual parameters are even compatible with array formal parameters (p. 80).

Assignment also meets the requirement (pp. 157-158).

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.P

SPL/I
Requirement R8

No implicit conversions	T
Explicit between integer, fixed point, and floating point	P
Explicit between fixed point scale factors	F
(Explicit between integer and Boolean)	T
(Explicit between integer and enumerated types)	F
(Explicit between different enumerated types)	F

SPL/I has no fixed point. However, conversions between integer and floating point, and floating point and fraction, are provided. (pp. B-22 through B-24). Many other explicit conversions are also provided, including between BIT and Boolean, and BIT and integer, and character string and integer, floating point, and fraction (pp. B-22 through B-32).

There are no implicit conversions - not even mixed mode arithmetic (pp. 87-103).

The necessary change is to add fixed point, see A2.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.P-

Implicit conversion between rangesF
Exception condition on integer and fixed point truncationP

The only instance of range in the language is integer and double precision integer, and as these are considered different types, explicit conversion is required (p. 102).

Run-time exception conditions on truncation are mentioned only for STRING and BIT assignment (p. 159).

Providing for run-time exception conditions on truncation would be of moderate to high difficulty; the language has no mechanisms in this area at all.

Adding ranges (and implicit conversions) would be of moderate difficulty.

B10. The base language will provide operations allowing

programs to interact with files, channels, or devices, including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

....P

Sending and receiving of data	T
Sending and receiving of control information	T
Dynamic device assignment	F
User exception condition control	F
Installation independence	F
(Data formatting capability)	F
(Reading and writing of bit strings)	I

There are only six I/O procedures in SPL/I: OPEN, CLOSE, PUT, GET, WRITELINE, and READLINE. They operate with virtual channels, and are installation and device dependent. Dynamic device reassignment is not provided. (pp. R-16 through R-21)

Just what the Tinman requires is not at all clear enough in detail to determine whether the necessary changes are fairly easy or moderately difficult.

R11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

....P

Union	T
Intersection	T
Difference	F
Complement	T
Membership predicate	F
(Set inclusion)	F

SPL/I has no powersets, but some of the required operations are provided on bit strings, which are in effect powersets where the elements are unknown to the compiler. (pp. 90-95)

It would be easy to add a difference operation. Adding a membership operation would require also adding powersets, which would take somewhat more work (but less than adding fixed point, for example).

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.F

Side effects must occur in left-to-right orderF
(Embedded assignments)F

The SPL/I rule is that the order of evaluation of operands of a single infix operator is undefined. This is a much better rule than the Tironian requirement, but it does not meet the requirement. (p. 27)

The change is moderately difficult. It would have a major effect on code generation in the compiler, and would lead to worse code in a number of cases.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.F†

Few precedence levelsT
No user-defined precedence levelsI
Operands of an operation are obviousP

There are nine precedence levels, which seems reasonable (p. 58). There is no way to define new operators, much less new precedence.

Operands of an operation are usually obvious, but A/B*C and A/B/C are permitted (p. 100).

Disallowing the two noted cases should be relatively easy.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.T

Fully met (pp. 87-129).

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.Pt

In a few contexts, only integer literals are allowed -- for example, the repetition factor within a structured constant (p. 111), and the labels on parts of a case statement (p. 138). Array upper bounds can however be expressions, evaluable at compile time (p. 64).

The required change would be fairly easy, although some care would be necessary to avoid syntactic ambiguity.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handling, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contribute to ease of learning.T

Parameter rules consistent in all contextsT
No special operations applicable only to parametersT

SPL/I really uses parameters only for procedure calls, so the question of consistency does not arise. Moreover, a procedure name cannot be passed as a parameter (pp. 51, 67, 79, 154), so no additional ways of binding arise for parameters.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.Pt

Actual and formal parameters will agree in typeT
Rank of parameter arrays is fixed at compile timeT
Parameter array size and subscript range can be passedPt

The only violation is that lower subscript bounds cannot be passed, as they are fixed at 1 by the language (p. 110). Note, however, that an array slice (pp. 118-121) can be passed as an array, to get some of the capability.

Binding of formal to actual parameters is discussed on pp. 79-81.

The necessary change is part of the work of adding lower bounds other than 1; see the comments on requirement A6.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.P

Act as constants (call by value plus)	T
Act as variables (call by reference)	T
Exception control	F
Procedure parameters	F
(Act as variables, but call by value)	T
(Act as variables, result parameter)	F

Both VALUE and INPUT parameters are provided. The first is the same as ALGOL 60 (the actual parameter does not change, but the formal parameter may); the second is the constant parameter called for in the Tinman. Call by reference (renaming) parameters are called INOUT.

There are no special provisions for exception control parameters, and procedure names cannot be parameters. (pp. 51, 67, 79-81, 154).

The required extensions would be moderately difficult, about the same amount of work as adding enumeration types. The missing features do not interact strongly with other language features.

C8. Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.F

Above properties optional	P-
Above properties are fixed at run time	F

Everything that can be specified about a formal parameter -- i.e., its mode and size -- must be specified. (pp. 67, 79). The concept of generic procedures is not in the language.

Note that if an array is passed through two levels of procedures, its size must be connected to the array at each level, even if the array is referenced only as an actual parameter at that level.

The addition of generic procedures would be fairly hard, particularly if efficient code is desired.

c9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

....P-

Variable number of arguments possibleP-
All but a constant number of arguments have the same typeF
Number of arguments in each call is fixed at compile timeT

Only the built-in GET and PUT I/O procedures accept a variable number of arguments, and nothing is said about type. (pp. 79-81, Appendix E)

The change would be at least moderately difficult.

D1. The user will have the ability to associate constant values of any type with identifiers.F

There is no way to associate an identifier with a constant in SPL/I. (p. 49)

The required change is trivial.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).P+

Literals for all built-in typesT
Consistent interpretation in program and dataU

Literals are provided not only for all built-in data types, but for generated types (e.g., arrays, records) as well, in a way which is easy to read. (pp. 107-113)

Nothing is said about consistency of interpretation for programs and data, but since precision is implementation dependent (p. 5?), we doubt that the requirement would be met by cross-compilers.

Providing precision in a machine independent way would be moderately difficult. See also #3.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.P+

Initial value can be specified as part of the declarationT
Initialization occurs at allocation scope entryT
No default initial valuesU

The requirement is probably fully met, but nothing is said about default initial values, and reference to a variable without a value is apparently not detected at run time. (See pp. 69-71, 114-123.)

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

....F

Numeric variable range specification mandatoryF
Fixed point variable step size specification mandatoryF
Range and step size specifications do not define a new typeF

SPL/I has no way to declare range or step size, and integer and double precision integer are treated as two different types, with explicit conversion necessary to assign one to the other. (See variable declarations, pp. 67-71; primitive modes, pp. 51-54; and assignment, pp. 157-158).

The necessary change is moderately difficult. The syntax needs to be changed to allow for range and step size declarations, and compile-time and run-time checks must be implemented. There is a strong interaction between this and the addition of fixed point.

D5. The range of values which can be associated with a variable, array, or record component, will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

....P+

Ranges of an enumeration type are allowedF
No arbitrary restrictions on the structure of dataT

There are no enumeration types, ergo no ranges thereof (pp. 49-68).

There are no arbitrary restrictions on the structure of data; records can be components of arrays and vice versa (pp. 49, 62-66). It is stated on page 117 that an array element is a scalar, but this is clearly a simple wording error in the manual.

Once enumeration types are added (see E6), and integer ranges are added (see D4), providing ranges of enumeration types would be trivial.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

....F

Recursive and network structures providedT
Handles variable-value and structure-component connectionsF
Pointer property is an attribute of a typed variableI
Pointer property not for constants, affects only assignmentP
Pointer property mandatory for dynamic allocationN/A
Allocation scope never wider than access scopeI
(Either the value or the pointer is modifiable)I
(Pointer mechanism handles procedures and parameters)F
(Remap and replace assignment have different syntaxes)I
(Built-in dynamic variable creation)I
(Variable equivalence classes are declarable)F

SPL/I's pointer mechanism is almost identical to that of Pascal; in other words, it is a good one. `POINTIP` is a mode, but pointer values are not directly accessible. A pointer variable points to an object of a specified type. Either the pointer or the pointed-to value can be assigned to. The latter is referenced as, e.g., `p6` (similar to Pascal's `p^n`). A pointer can share its pointed-to value only with another pointer variable. A heap allocation mechanism is provided. (pp. 59-61, 128-129, 9-10, B-11)

To change SPL/I's pointer mechanism to meet the Tinman requirement would be a lot of work. Much of the work would be in defining the requirement itself to be a viable, sensible one, in more detail than at present.

E1. The user of the language will be able to define new data types and operations within programs.

....P-

No special mechanisms are provided for defining new types. However, arrays and records are provided as type generators, and own variables (called STATIC) would allow one to define and use a new type via procedure calls, with the components of the data objects inaccessible except via the procedures. This provides much of the required capability, albeit inefficiently, even though the methods are perhaps different from those envisioned by the Tinman.

Addition of a special purpose abstract type mechanism would be a major effort.

E2. The "use" of defined types will be indistinguishable from built-in types.

....F

See the comments on requirement E1.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

....T

Fully met (see pp. 36-40).

E4. The user will be able, within the source language, to extend existing operators to new data types.

....F

See the comments on requirement E1.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the

type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

....P-

Construction	T
Selection	F
Predicates	F
Type conversions	T
Operations and data can be defined together	P

The only way to define operations on new types is by procedures. This allows construction and type conversion operations, with a notation much like that for built-in types. It does not allow the definition of infix predicate operators, or of selection operators which can appear on the left hand side of an assignment statement.

See the comments on requirement E1; a major change would be required.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the powerset of an enumeration type. These definitions will be processed entirely at compile time.

....P

Enumeration	F
Cartesian products (records)	T
Discriminated union	F
Powerset of an enumeration type	F

Only pointers, records, and arrays are provided as type generators. (pp. 58-66)

A moderate change would be required. Adding enumeration types and powersets would not be too hard; most powerset operators are already provided (see R-11). Adding discriminated union would be somewhat harder, as it would affect the syntax and semantics of more statements (e.g., case).

E7. Type definitions by free union (i.e., union of

non-disjoint types) and subsetting are not desired.1

Only pointers, records and arrays are provided as type generators.
(cp. 5°-66)

FR. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.F-

Initialization
Finalization
Allocation actions
Deallocation actions

Allocation could be defined as a procedure, using either a large pre-declared array or heap storage via pointers. Initialization could also be defined as a procedure, but with a notation considerably different from that for built-in types. Finalization and deallocation cannot be defined in a sensible way, because there is no way for a procedure to automatically get control at exit from a scope.

See the comments on requirement E1.

F1. The language will allow the user to distinguish between scope of allocation and scope of access.P

Allocation and access scope can be different, but only if the allocation scope is the whole program, as in ALLOC 60 or CS-4 (pp. 69-71).

The change is fairly easy. The AUTOMATIC storage class declaration would need an optional parameter, giving the block name of the allocation scope block.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.P

Allowable operations can be limited1
Access can be limited where used1
External declarations need not all have the same scope1
Naming conflicts can be avoided (renaming)F

Access can be limited where a structure is used by only declaring as EXTERNAL the desired structures. There is, however, only a single external scope (GLOBAL on definition, EXTERNAL on reference). Allowable operations can be limited by making the data objects own (STATIC) within a procedure defining the operations. There are no re-naming provisions. (pp. 69-71)

Satisfying the requirement with a special purpose "cluster" mechanism would be a fairly major change, closely connected to that required by E1.

F3. The scope of identifiers will be wholly determined at compile time.1

Fully met (pp. 36-40).

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

....F

The run-time support library contains no such procedure (Appendix 3).

Once abstract types are added (requirement F1), then meeting this requirement could mean only a little work or a tremendous amount, depending on the size of the desired library.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

....P-

Program component libraries accessible at compile timeF
Libraries can contain foreign language routinesF
Interface requirements checkable at compile timeT

Compile-time libraries are not mentioned in the Reference Manual. It is, however, possible to declare the properties of separate compilation modules. (pp. 33-35)

The necessary changes are at least moderate in scope.

F6. Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

....U

Libraries and compools will be indistinguishableU
Immediately accessible sublibraries at any levelU

There is no mention of library mechanisms in the Reference Manual. (The requirement is not really a language requirement).

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.F

while the multiprocessing and signalling statements (pp. 159-196) might be usable to partly meet this requirement, their connection to real hardware is implementation dependent (p. 187).

The requirement is so vague that it is impossible to estimate the scope of the necessary change.

AD-A037 640 COMPUTER SCIENCES CORP FALLS CHURCH VA
DOD PROGRAM FOR SOFTWARE COMMONALITY HIGH ORDER LANGUAGE WORKIN--ETC(U)
1977 F/G 9/2
N00039-75-C-0289
NL

UNCLASSIFIED

5 OF 5
ADA037640



END

DATE
FILMED
4-77

61. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.P+

Sequential executionT
Conditional executionT
IterationT
RecursionT
(Pseudo) parallel processingT
Exception handlingP-
Asynchronous interrupt handlingP-
Control structures from a small set of simple primitivesT

While the parallel processing facilities of the language could presumably be made to handle interrupts, the association of hardware signals with software signal variables is not part of the language definition (p. 186). (The language does specify a syntax for associating both hardware and software signals with signal variables, but the semantics of the association are implementation-dependent.)

There are no mechanisms for handling specific exceptions (the same implementation-dependent technique mentioned in the previous paragraph is available, but each implementation will specify which exceptions can be handled), and neither a label nor a procedure name can be passed as a parameter (pp. 90-104, 124, 126, 137).

Control structures are described in pp. 131-196.

Only a moderate change would be required. Much of the necessary mechanism is already present in the parallel processing facilities.

62. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.I

Fully met. (p. 156)

While the requirement is met, the restrictions on GOTO and on ENDLOOP, EXIT, and ENDITERATION statements will lead to very awkward programming in some cases. For example, ENDLOOP cannot lead to outside of a BEGIN block in the iteration statement; EXIT must be used (and in most cases, ENDLOOP later). (pp. 147-150)

An unrestricted GOTO, used carefully, would lead to more readable, more efficient programs.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.P+

Based on Boolean expressionT
Based on type from discriminated unionF
Based on computed choice among labeled alternativesT
All alternative must be accounted forT
Simple mechanisms will be supplied for common casesT

The conditional control structures are IF...THEN...ELSE...ENDIF and DO CASE...OF...ELSE...ENDCASE (pp. 136-139). They meet the requirements except in relation to discriminated union, which the language does not have.

Once discriminated union is added, providing conditional control structures based on it will be easy.

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).P

Termination can occur anywhere in the loopT
Multiple terminating predicates are possibleT
Entry permitted only at the loop headT
Simple cases are clear and efficientP
Control variable is local to the loopF
Control value is efficiently available after terminationP

SPL/I has fairly clean iterative statements: WHILE, UNTIL, and FOR (pp. 140-146). The control expressions for the FOR are evaluated at the start; better efficiency might result from a restriction against

modifying their values in the loop. The value of the control variable is always available on loop termination, so it is not local to the loop, and this will not be efficient in all cases (e.g., where the user didn't need the value). Another FOR inefficiency is mentioned under J1.

The necessary changes are straightforward, and would probably be fairly easy, but not trivial, to make.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.F

No recursive procedures within recursive proceduresF
(Maximum depth of recursion can be specified)F
(Recursiveness must be specified)P

There is no restriction against nesting of recursive procedures (pp. 76-77, 41-44). It is possible to declare a procedure to be NONREENTRANT, which can greatly increase efficiency.

Adding the restriction would be trivial; changing the compiler to take advantage of the extra information to produce more efficient code would probably be somewhat harder.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.P

Able to create and terminate parallel processesT
Process can gain exclusive use of resourcesT
No parallel routines within recursive routinesT
No routines within parallel routinesF
Maximum number of simultaneous instances are declarableF
(Access rules are enforced)F

Parallel processing facilities are described on pp. 159-196; they appear overly complex for the capability provided, and they clearly depend on an (expensive) heap storage mechanism. A parallel module must be a program declaration (p. 159), which cannot be contained within a

procedure declaration, but can contain another program declaration (pp. 74-75).

Meeting the letter of the requirement would not be hard. Changing the mechanism to be simpler and more efficient would however be a major design task.

G7. The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.F

Program can get control for any exceptionF
Parameters can be passedF
Can get out of any level of a nest of controlF
Can handle the exception at any level of controlF

The language has no such facility.

A moderate to major change would be required.

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.P-

Priority specificationF
Synchronization via wait/enable operationsT
Wait for end of real time intervalF
Wait for end of simulated time intervalF
Wait for hardware interruptP
(Can enable and disable interrupts)T

There is no direct provision for specification of priorities among parallel paths, although in special situations the counts on signal variables could be used to control priority. The WAIT (p. 195) could in principle be used to wait for a time interval or a hardware interrupt, but there is no language-defined connection between SIGNAL variables and hardware (p. 185). (See the comments on exception handling, requirement G1.)

In this reviewer's opinion, the parallel processing feature of SPL/I should be totally redesigned -- a major task. Just making the modifications explicitly required here would be of moderate difficulty.

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.T

Free format with statement terminatorT
Mnemonic identifiers possibleT
Based on conventional formsT
Simple grammarT
No special case notationsT
No abbreviations of identifiers or keywordsT
Unambiguous grammarT

SPL/I has a straightforward, mostly clean syntax. For example, IF, WHILE, UNTIL, CASE, and FOR statements must have a corresponding ending bracket (e.g., ENDIF), so that BEGIN...END brackets are unnecessary. For the most part, someone unfamiliar with SPL/I can understand programs in the language.

One unfortunate rule is that a statement may have only one label (p. 47); thus multiple labels must be written L1:; L2:; L3: statement.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.T

There are no such facilities in SPL/I.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.T

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.T

Break character existsT
(Literals are self-identifying as to type)T
(Bit-string literals for any type)F

The SPL/I break character for identifiers is the underscore (p. 7). Literals for both primitive and structured types are self-identifying as to type (pp. 108, 111-113).

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.T

See page 12.

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.T

There are 82 reserved words, which seems reasonable, if not outstanding (p. A-5).

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.P+

Uniform comment conventionT
Look different from codeT

Bracketed by one or two charactersT
Can contain any charactersP
Can appear anywhere reasonableT
Terminated by the end of the lineU
Compatible with automatic reformattingT

Comments are enclosed in square brackets; they may contain any characters, except that included square brackets must be nested (p. 32). They can appear between any tokens (p. 30). The manual does not say whether end of line terminates a comment; it seems unlikely.

A trivial change would be required to fully meet this requirement.

H8. The language will not permit unmatched parentheses of any kind.T

See entire syntax; all brackets must be in pairs.

H9. There will be a uniform referent notation.T

Fully met. See pp. 116, 124-126.

H10. No language defined symbols appearing in the same context will have essentially different meanings.T

11. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.F

SPL/I is full of implementation dependent defaults -- for example, the range and precision of variables (pp. 51-52) and of counters on SIGNAL variables (n. 53), and the association of hardware signals with signal identifiers (p. 186). In addition, the association of a system resource with a resource variable and of a software signal source with a signal identifier are made by programmer conventions (p. 171, 186).

Meeting the requirement would be of at least medium-level difficulty. Not only range declarations, etc., would be required, but also the ranges of arithmetic results would have to be precisely defined in terms of the ranges of their operands.

12. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.P+

Defaults specified for don't care casesT
Programmer can override the defaultsP

Such things as data representation, parameter class (p. 80), storage class (p. 69), reentrancy (n. 76-77), and closed calls are decided by default if left unspecified. Not all can be overridden -- in particular, data representation or closed calls.

Open calls would not be hard to provide, but allowing programmer control over data representation would entail at least a moderate amount of work.

13. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.F

SPL/I has no such facility.

The necessary addition would be fairly easy, if done minimally.

14. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.F

SPL/I has no such facility.

The required addition would not be trivial, but not very hard either.

15. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.F

The whole language is a unit; there is no kernel on which the rest is based.

To define and extract such a kernel would be a major change.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.F

The maximum number of characters in a name is part of the language definition (it is 16 -- see p. 16). Limits on the number of array dimensions, the size of each (p. 65), the number of variables (see, e.g., p. 175), and depth of parentheses nesting in expressions are all implementation dependent.

The additional specification should be easy; the effects on existing compilers might not be.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.T

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.P

No efficiency cost for unused featuresP
Efficient code can be produced for all featuresP

Some inefficiencies in SPL/I are:

1. The direction of a FOR loop is dynamically determined according to the sign of the value of the BY expression; this is unnecessary, and inefficient. Other FOR inefficiencies are mentioned under G4.
2. Pointed-to values can only be in the heap; often more efficient storage discipline would suffice, but SPL/I does not provide for them for these objects (p. 59).
3. A scalar actual parameter can match an array formal parameter (p. 80). We suspect this raises the cost even for the passage of normal arrays as parameters.
4. The rules for GOTO surrogates cause inefficiency.

SPL/I does have a side effect rule which allows efficient code, and allows declaration of non-recursiveness for procedures (both contrary to the Timman).

At least a moderate effort would be required. The FOR rules would not be hard to change, but providing a choice of storage discipline for pointed-to values, and a good set of GOTO surrogates would be harder (unless the restrictions on GOTO were simply removed, as we believe they should be).

J2. Any optimizations performed by the translator will not change the effect of the program.U

This is not a language requirement.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.F

SPL/I has no such capability.

The necessary change could be easy to moderately difficult, depending on the degree of integration of SPL/I code and machine code, the protections against loss of optimization, etc.

J4. It will be possible within the source language to specify the object representation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.F

Encapsulated specification of representation possibleF
Space/time tradeoff can be specifiedF

SPL/I has no such capability

A moderate to major change is required.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.F

Open/closed properties can be specifiedF
Open and closed versions have the same semanticsF

SPL/I has no such facility; all procedures are closed.

A moderate change is required.

Extraneous Features

We recommend that the following features of SPL/I, not required by the Tinman, be kept:

- * CFLOAT (complex) type.
- * STRING type (assuming that the Tinman requires only a character type, not a character string type).
- * SIGNAL, PFSOURCE, and PROCESS types. Although not explicitly called for by the Tinman as types, they are needed in the SPL/I realization of other Tinman requirements (parallel processing). They should be redesigned.
- * Call by value parameters.
- * Bit string literals.

We recommend that the following features of SPL/I, not required by the Tinman, be deleted:

- * FRAC (fraction) type. The use of this type is not clear and it can probably be replaced by a fixed-point type.
- * DINT (double precision integer) type. The functionality obtained with this type can be assumed by range specifications on integer type.
- * CFRAC (complex fraction) and CINT (complex integer) types. These appear to be of extremely limited usefulness.
- * Array slices.

It is recommended that the CFLOAT type be retained because it is a numeric type and the user should be able to use the numeric operators with data of that type. If, however, SPL/I is extended to permit extension of built-in operators to user-defined types (which are themselves extensions to SPL/I), then CFLOAT could be deleted from the base language.

Summary

Despite the fact that it was designed for signal processing applications, SPL/I is a general purpose language, and it is a fairly clean one, with the exception of the parallel processing features. The last may be an unfair criticism, because we have seen no language which provides all necessary mechanisms for parallel processing cleanly and efficiently -- maybe it cannot be done (and therefore should not be a Tinman requirement). Still, we believe it could be done better than in SPL/I.

In many respects SPL/I conforms to the spirit of the Tinman requirements; for example, it has:

- * Strong type-checking, without implicit conversions.
- * Powerful data structuring tools (without discriminated union or alternate record structures, however).
- * Support for parallel processing.
- * Structured control mechanisms (although there are some problems with individual mechanisms, as the detailed evaluation shows).
- * Variable initialization.
- * Attention to separate compilation.
- * Recursive procedures, with the ability to declare non-recursiveness.
- * Pointer variables, in a way similar to Pascal -- i.e., a clean way.
- * Consistent treatment of structures. For example, a function result can be structured, and structured literals can be built.

SPL/I is also a reasonable-sized (not too big), fairly simple, language. It pays for this, however, in missing features. For example, the following are not provided:

- * Range and precision specification for variables.
- * Powersets, and enumeration types as such.
- * Union of types and alternate record structures.
- * The ability to define new types, including operations on objects of the type.

- * Encapsulated machine code.
- * Control over representation of data objects.
- * Dynamic upper array bound, or lower bound different from 1.
- * Fixed point type.
- * Identifiers to stand for constants.
- * Conditional compilation.

On balance, SPL/I is a much cleaner, more readable, simpler language, than, for example, CS-4. It even has two very important features that CS-4 does not: Recursive procedures and pointers. It is missing a number of features in CS-4, but most of these are provided in such an awkward way in CS-4 that extending SPL/I would probably result in a cleaner language.

On the other hand, we see little reason to chose SPL/I instead of Pascal as a basis for extension to meet the final version of the Tinman requirements. The chief features in SPL/I which are not in Pascal are parallel processing, structured function results, variable initialization, and multiple storage classes -- and indeed, all but the first are done well. However, these features could easily be add to Pascal. Pascal has, and SPL/I does not, powersets, enumeration types, ranges of values, and alternate record structures. Pascal also has better (more efficient, safer) iteration control structures, and is probably a smaller, simpler language (easier to read and compile) than SPL/I. In addition, Pascal is much better known, and some of the best features of SPL/I have been taken directly from Pascal (e.g., the method of providing pointers).

The SPL/I Language Reference Manual is fairly good -- it is consistently and carefully organized and defines most of the features of the language. However, it could be improved greatly by the addition of more interesting examples (the given examples never provide the answer to a questionable point in the description), the insertion of more summary descriptions (as it is, the reader often has to refer to many different pages to find the answer to a simple question), and an expanded index.